版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Broadview
www.broadview.com.cn

[PACKT] open source*
PUBLISHING
community experience distilled



Scala High Performance Programming

高性能Scala

用Scala和函数式编程范式开发高性能软件

[美] Vincent Theron Michael Diamant 著
杨云 廖光明 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

[PACKT] open source
PUBLISHING community experience distilled

Scala High Performance Programming

高性能Scala

用Scala和函数式编程范式开发高性能软件

[美] Vincent Theron Michael Diamant 著
杨云 廖光明 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Scala 是一种表达能力非常强的语言，能够用非常简洁的代码表达丰富的业务含义。为了在生产上充分发挥 Scala 的能力，除了掌握其简洁的语法外，理解 Scala 在性能上的特点和优化点也是非常重要的事。

本书通过解析一个金融领域高频交易的实际例子，引领读者掌握如何对 Scala 程序（以及一般 JVM 程序）进行基准测试和性能分析，从而找出瓶颈。随后作者介绍了 Scala 语言、Scala 标准库（尤其是集合库）以及 Scalaz 库里解决相应瓶颈的各种技巧。并行计算和分布式架构作为性能调优的重要手段，更是本书的重中之重，作者对 Scala 的并行计算和分布式架构中存在的问题都进行了充分的讨论和讲解，非常值得学习。

本书适合已经具有 Scala 编程基础、能够较好地使用 Scala 风格代码实现业务功能的程序员，作为在性能优化方面的深造阅读资料。

Copyright©2016 Packt Publishing.

First published in the English language under the title 'Scala High Performance Programming'.

本书简体中文版专有出版权由 Packt Publishing 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号　图字：01-2016-7237

图书在版编目（CIP）数据

高性能 Scala /（美）文森特·西伦（Vincent Theron），（美）迈克尔·迪亚芒（Michael Diamant）著；杨云，廖光明译. —北京：电子工业出版社，2017.5

书名原文：Scala High Performance Programming

ISBN 978-7-121-31237-3

Ⅰ. ①高… Ⅱ. ①文… ②迈… ③杨… ④廖… Ⅲ.①JAVA 语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2017)第 065972 号

责任编辑：张春雨

印　　刷：三河市良远印务有限公司

装　　订：三河市良远印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱　邮编：100036

开　　本：787×980　1/16　印张：15.75　字数：352.8 千字

版　　次：2017 年 5 月第 1 版

印　　次：2017 年 5 月第 1 次印刷

定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

关于作者

Vincent Theron 是一个拥有 9 年工作经验的专业软件工程师。他在 6 年前发现了 Scala 并将之应用于构建高伸缩性、高可靠的应用。他在多个行业设计软件来解决商业上的问题，包括在线博彩、金融交易，以及最近的广告行业。在巴黎东部 Marne-la-Vallée，Vincent 获得了计算机科学及软件工程硕士学位。他和妻子、孩子以及两只毛茸茸的猫一起在波士顿区域生活。

感谢在 Packt 出版社的每一个人，是你们的努力工作才能让这本书面世。感谢柴塔尼亚-耐尔，带着这个出书项目找到我。感谢 Nikhil Borkar，一路上给我提供指引。感谢 Michael Diamant，我的合作作者，我的同事，同时也是朋友，感谢你给这本书带来的知识以及充满灵感的每一天。感谢我的父母，感谢你们的爱和支持，以及你们给我买的第一台电脑。最后，感谢我的妻子，Julie，感谢你一直的鼓励，感谢你给了我一个这么绝妙的儿子。

Michael Diamant 是一个专业的软件工程师，热衷于函数式编程。他在 2009 年开始自己的事业，专注于 Java 和面向对象的编程范式。在 2011 年学习 Scala 之后，他专注于使用 Scala 和函数式编程范式来构建金融交易和广告领域的软件系统。迈克尔毕业于伍斯特理工学院，居住在波士顿区域。

我能在这本书里面分享的知识来源于他人在我一生中给予的支持和教导。我想要特别感谢我的同事文森特，感谢你推动我付出这些努力，感谢所有在一起的时间，让我们可以发展这些书中的想法。我所有的现在和之前的同事都帮助我提高了工程技术，没有你们慷慨地分享你们的所学，我将不可能编写这本书。除了 Vincent，我还想特别提及一些我觉得尤其想感谢的同事：Dave Stevens、Gary Malouf、Eugene Kolnick 和 Johnny Everson。感谢我的父母和兄弟，你们支持我，将我培养成现在的我。我尤其想深深地感谢我的女友 Anna 在整个写书过程中给予我的支持。最后，同样的感谢送给 Packt 出版社，在你们的帮助下我们写了我们的第一本书。

关于审阅者

Nermin Šerifović 是一个从 2009 年开始就热衷 Scala 的人，他从 2011 年开始在专业领域应用 Scala。在大部分工作生涯中，他专注于使用 JVM 技术构建后端平台。最近，作为 Pingup 的研发总监，他正在一个本地化服务预订系统上领导开发工作。

Nermin 是一个哈佛大学延伸教育学院的导师，他在那里与人合作教授 Scala 并行编程的课程，他还在多个学术会议上发表了演讲。

作为一个 Scala 社区的活跃会员，Nermin 组织了波士顿区域的 Scala 爱好者群，同时他也是中东 Scala 座谈基金会一员。他是 *Scala Puzzlers* 一书的合作作者，Scala 解惑网站的合作创建者。

Nermin 拥有康奈尔大学计算机科学硕士学位，他感兴趣的领域包括分布式系统及伴随着的并行，响应式和函数式编程。

目录

前言 ix

1 高性能之路 1

性能的定义 2

高性能软件 2

硬件资源 3

时延和吞吐率 4

瓶颈 5

性能总结 5

平均数的问题 6

百分位数来救场 8

指标搜集 9

用基准数据（benchmark）来衡量性能 9

通过 Profiling 来定位瓶颈 10

结合基准测试和 profiling 10

案例分析 11

工具链 11

小结 12

2 在 JVM 上度量性能 13

金融领域一瞥 13

意外的市场波动毁掉了利润 16

重现故障 17

吞吐量基准测试 17

时延基准测试 20
定位瓶颈 25
微基准取得大进步 42
小结 49

3 释放 Scala 的性能 51

值类 52
字节码表示 52
性能考虑 54
标记类型——值类的一种替代品 55
专门化 57
字节码表示 58
性能考虑 60
元组 65
字节码表示 65
性能考虑 66
模式匹配 68
字节码表示 68
性能考虑 70
尾递归 75
字节码表示 78
性能考虑 79
Option 数据类型 83
字节码表示 83
性能考虑 84
案例研究——性能更高的 Option 85
小结 89

4 探索集合 API 91

高吞吐量系统 - 改进指令簿 91

理解过去实现上的折中 - list 实现 92
当前的指令簿 - queue 实现 101
通过惰性计算来提升取消操作的性能 104
历史数据分析 114
滞后时序收益率（lagged time series returns） 114
处理多个收益率序列 122
小结 127

5 惰性集合及事件溯源 129

提升用户报表生成速度 129
深入报表生成代码 130
使用视图提速报表生成 133
视图的注意事项 141
打包报表生成结果 145
重新思考报表架构 146
Stream 概览 149
事件变换 152
构建事件源管道 158
马尔可夫流式链 162
流的注意事项 166
小结 169

6 Scala 的并发 171

并行回测（backtesting）策略 171
探索 Future 173
Future 和 crazy ideas 177
Future 使用时的考量 179
提交执行妨碍性能 185
处理阻塞调用和回调 188
ExecutionContext 和阻塞调用 189

用 Promise 转化回调 193
受命进一步提升回测性能 196
介绍 Scalaz Task 197
用 Task 为交易日模拟建模 204
总结回测 209
小结 210

7 高性能架构 211

分布式自动化交易员（Distributed automated traders） 211
分布式架构概述 212
第一次尝试分布式自动化交易系统 212
引入 CRDT 214
CRDT 和自动化交易系统 219
当余额不足时 220
免费交易策略性能提升 222
为交易策略做基准测试 222
无界队列（unbounded queue）的危险 225
应用背压（back pressure） 226
应用负载控制策略 227
Free monad 233
小结 240

前言

Scala 是一个大胆的（audacious）编程语言，在 JVM 平台上，它将面向对象和函数式编程的概念融合到一起。Scala 从一个相对小众的语言成长为了一个开发健壮且可维护的 JVM 应用的首选语言。然而，如果不深入了解这门语言和它提供的高级特性，编写高性能应用依然是一个挑战性的任务。

从 2011 年开始，我们就使用 Scala 来解决复杂的业务上的挑战，这些挑战常常有很高的性能要求。在本书中，我们分享了在这些年中所学到的东西及在编写软件中应用过的技术。我们在书中浏览了这门语言及其生态系统，包括周边工具和使用广泛的库。

我们编写这本书的目标是帮助你理解 Scala 语言提供给你的选择。你有权在书中收集必要的信息，以便在你自己的软件系统中做出更有知识基础的设计和实现决定。我们将不仅给你 Scala 之鱼，带你上路，还将传授渔鱼之技，并给你相关工具让你写出更函数式性能更高的软件。一路上，我们将通过精心设计的业务问题来激发技术讨论，这些业务问题将能让你回想起真实世界的问题。希望通过阅读本书，你能体会到 Scala 的强大能力，并找到合适的工具去编写函数式性能更好的应用。

本书的内容包含什么

第 1 章，*通往性能之路*，介绍了性能的概念及关于这个主题的一些重要术语。

第 2 章，*在 JVM 上衡量性能*，详细介绍了 JVM 上可用于衡量和评估性能的工具，包括 JMH 和 Flight Recorder。

第 3 章，*释放 Scala 的性能*，对多项利用 Scala 语言特性来提升程序性能的技术和模式提供了引导。

第 4 章，*探索集合 API*，讨论了 Scala 标准库提供的几个集合抽象。在这一章中我们专注于急切执行的集合。

第 5 章，惰性集合和事件溯源，是一个高级技术章节，讨论了两类延迟序列：视图和流。我们还对事件源范式进行了简要的介绍。

第 6 章，Scala 的并发，讨论了编写健壮的并行代码的重要性。我们深入介绍了 Scala 标准库提供的 Future API，并介绍了 Scalaz 库中提出的任务抽象。

第 7 章，高性能架构，作为最后一章，覆盖了前面章节讲过的主题的更深入的相关知识，并探索了使用 CRDT 作为分布式系统的构建块。这一章还探索了在面临高吞吐量时，使用 free monald 实现负载控制策略，以构建响应延时可控的系统。

你需要准备什么

你应该要在你的操作系统上安装好 Java Development Kit 8 版本或更高，以便你可以运行所有的示例代码。本书讨论了 Oracle HotSpot JVM，并演示了在 Oracle JDK 中包含的工具。你应该还需要从 `http://www.scala-sbt.org/download.html` 获取最新版的 sbt（在写作本书的时候，版本为 0.13.11）。

这本书为谁而写

你应该具备关于 Scala 编程语言的基本知识，熟悉一些基本的函数式编程概念，并具有编写产品级 JVM 软件的经验。我们建议刚接触 Scala 和函数式编程的读者在读本书之前花一些时间学习其他的资源，以便可以从书中获取到更多的知识。两个杰出的以 Scala 为中心的资源是 Artima 出版社的《Scala 编程》（*Programming in Scala*）及 Manning 出版社的《Scala 函数式编程》（*Functional Programming in Scala*）。前一个最适用于具有坚实的面向对象 Java 编程知识，同时想要先理解这门语言然后理解函数式编程范式的开发者。后者则聚焦于函数式编程范式而少于讨论某一个具体语言中的概念。

约定

在本书中，你将能找到大量的用于区分不同类型信息的文本样式。下面是一些示例及其所表示的意义。文本中的代码、数据库表名、文件夹名、文件名、文件扩展名、路径名、示例 URL 及用户输入将这样显示：“`-XX:+FlightRecorderOptions` 接受一个名为

settings 的参数，此参数默认值为$JAVA_HOME/jre/lib/jfr/default.jfc."。

代码块格式如下：

```
def sum(l: List[Int]): Int = l match {
  case Nil => 0
  case x :: xs => x + sum(xs)
}
```

任意的命令行输入和输出格式如下：

```
    sbt 'project chapter2' 'set javaOptions := Seq("-Xmx1G")' 'runMain
highperfscala.benchmarks.ThroughputBenchmark
src/main/resources/historical_data 250000'
```

新的术语和重要的词汇**粗体**显示。在屏幕上面看到的文本，如菜单和对话框，显示为这样的文本样式：让我们从 **Code** 标签组中的 **Overview** 标签开始。



警告或者重要的说明像这样表示。



小建议及技巧像这样表示。

下载示例代码

你可以从 http://www.broadview.com.cn 的“下载专区”，下载所有已购买的博文视点书籍的示例代码文件。

勘误表

虽然我们已经尽力谨慎地确保内容的准确性，但错误仍然存在。如果你发现了书中的错误，包括正文和代码中的错误，请告诉我们，我们会非常感激。这样，你不仅帮助了其他读者，也帮助我们改进后续的出版。如发现任何勘误，可以在博文视点网站相应图书的页面提交勘误信息。一旦你找到的错误被证实，你提交的信息就会被接受，我们的网站也会发布这些勘误信息。你可以随时浏览图书页面，查看已发布的勘误信息。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），您即可享受以下服务：

- 下载资源：本书所提供的示例代码及资源文件均可在【下载资源】处下载。
- 提交勘误：您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- 与作者交流：在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31237

二维码：



1 高性能之路

欢迎加入学习旅程，我们将学会如何用实用的方式用 Scala 编程语言和函数式编程范式来编写高性能、高效率的软件。函数式编程概念，比如纯函数和高阶函数、引用透明和不变性（immutability）是非常理想的工程特性。我们可以用这些特性编写可组合的组件、可维护的软件，以及表达力强且容易理解的代码。但是，尽管有这么多优点，函数式编程经常被（错误地）和低性能、低效的代码联系在一起。本书的目的是告诉你并非如此！本书探索了如何利用函数式编程、Scala 编程语言特性、Scala 标准库和 Scala 生态圈来编写高性能的软件。

Scala 是一种静态强类型的编程语言，它视图优雅地把函数式编程和面向对象编程范式结合起来。最近几年 Scala 日渐流行，因为使用 Scala 你可以既优雅又实用地使用函数式编程范式来开发用于生产环境的软件。Scala 代码编译成字节码，运行在 Java 虚拟机（JVM）上，而 JVM 是一种广为接受的运行平台，可配置，而且提供了优秀的自检（introspect）和调试工具来排查程序错误和性能问题。Scala 的一个附带的好处就是它可以很好地和 Java 互操作，从而使你可以使用已有的 Java 库。尽管 Scala 编译器和 JVM 都在持续地优化，而且现在已经能生成优化良好的字节码，但要达成你的性能目标，仍然需要依靠作为开发者的你的努力。

在一头扎进 Scala 和 JVM 规范前，对于我们所追寻的圣杯——性能——我们先要建立统一的认识。在第 1 章，我们将讨论与编程语言无关的性能概念的基础。我们将提出和解释全书中将要使用的术语和概念。

本章着重讨论以下主题：

- 性能定义
- 性能总结
- 指标搜集

我们还将介绍一个研究案例，一个基于真实问题的虚构应用，这可以帮助我们展示后面章节的技巧和模式。

性能的定义

一个性能词汇表能帮助你限定你手头问题的类型范围，并且能经常指引你找到解决方案。尤其是在时间紧张的情况下，可靠的直觉和有序的策略是你解决性能问题的重要资源。

我们首先要对一个术语建立统一理解：性能。这个术语用于定性地或定量地评估一个软件达成其目标的能力。这个“目标”根据手头的情况，可能差异很大，但是作为专业的软件工程师，这个目标最终必须和业务目的相一致。和业务团队紧密合作来识别出业务上的性能指标是非常重要的。比如对于一个面向终端用户的电商网站，确认并发用户数和可接受的请求响应时间是关键。对于一个金融交易公司来说，交易延迟时间可能是最重要的竞争优势。同时还要注意其他的非功能指标，比如“交易不能丢失”这种行业规范和外部监管要求。这些业务领域约束也会影响你的软件的性能指标。建立明确且被一致认同的领域图景是关键第一步。如果无法明确定义这些约束，那么也无法得出一个有效的解决方案。



需求搜集本身是一个复杂的主题，超出了本书的范畴。如果你想深入这个主题，我们推荐 Gojko Adzic 的两本书：*Impact Mapping: Making a big impact with software products and projects*（http://www.amazon.com/Impa ct-Mapping-software-products-projects-ebook/dp/B009KWDKVA）和 *Fifty Quick Ideas to Improve Your User Stories*（http://www.amazon.com/Fifty-Quick-Ideas-Improve-Stories-ebook/dp/B00OGT2U7 M）。

高性能软件

设计高性能软件是我们作为软件工程师的目标之一。思考这个目标引致一个普遍被问

及的问题：多高的性能算够高？我们用高性能这个术语来描述达到最低可接受阈值的性能表现。我们的目标是达到——如果可能——超出最低可接受阈值。深思一下：如果对于可接受的性能没有统一的标准，显然就不可能产出高性能的软件！这充分说明在开发高性能软件之前，给出明确的性能定义的重要性。



花点时间来思考对你的业务领域来说高性能的定义是什么。你是否曾经为达到你的高性能定义而绞尽脑汁？考虑一下你为解决性能困境采取过的措施。哪些措施是有效的，哪些是无效的？在你阅读本书的过程中，请保持思考这些问题，以便考察哪些技巧能更有效地帮助你达到你的高性能定义。

硬件资源

为了为高性能软件定义指标，我们必须扩展我们的性能词汇表。首先，开始关注你的环境资源。我们用资源这个术语来代表软件运行的所有基础设置。参考下面的资源清单，其中列出了在投入性能调优练习前你需要搜集的全部资源信息：

- 硬件类型：物理或虚拟
- CPU：
 核数
 L1、L2 和 L3 缓存大小
 NUMA zone
- 内存（比如 16GB）
- 网络连接速率（比如 1GbE 或 10GbE）
- 操作系统和内核版本
- 内核设置（比如 TCP socket receive 缓存大小）
- JVM 版本

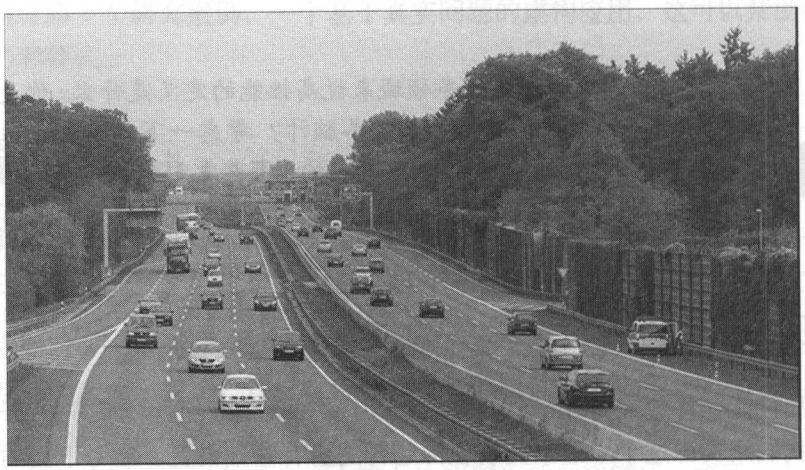
把这些资源清单详细列举可以迫使你思考运行环境的性能极限。



关于内核优化的优质资料包括 Red Hat 性能调优指南（https://goo.gl/gDS5mY）和 Brendan Gregg 的演讲材料和教程（http://www.brendangregg.com/linuxperf.html）。

时延和吞吐率

时延和吞吐率定义了两种不同类型的性能，经常用来建立性能指标。如下图（德国Autobahn）所示的高速公路图很适合作为比喻来帮助大家建立对这两种性能类型的直观认识：



高速公路的例子能帮助我们理解什么叫时延，什么叫吞吐量。

（图片来自wikimedia，https://en.wikipedia.org/wiki/Highway#/media/File:Blick_auf_A_2_bei_Raststt%C3%A4tte_Lehrter_See_(2009).jpg，图片遵循creative commons许可协议BY-SA 3.0）

时延描述一个被观察的流程完成所花费的时间。以高速公路例子来说，流程是一辆车沿着一条高速公路道行驶。如果高速公路不拥堵，那么车辆可以快速通行。这个流程就被称为低时延。如果高速公路很拥堵，那么通行时间会增加，被称为高时延或缓慢的流程。你能采取的性能优化措施也可以用这个高速公路来比喻。你可以想象把一个昂贵的多项式算法改写成线性算法就好比提高高速路的路面质量或者减少车辆的路面摩擦力。减少摩擦力使车子可以低时延地通过高速路。在实践中，时延性能目标经常用你的业务领域的最大可容忍时延来定义。

吞吐率定义为观察到的流程完成的频率。用高速路类比来说，在单位时间内从 A 点开到 B 点的车辆的数量是高速路的吞吐率。举例来说，如果有 3 条道，每辆车以统一的速率在每条道上行驶，那么吞吐率就是（在观察期内每条道上从 A 点开到 B 点的车数）* 3。归纳推理一下可能会注意到吞吐率和时延之间可能存在强的负相关性（这两个指标是冲突的）。也就是说时延提高时吞吐率下降。事实上，在不少案例中并非绝对如此。请牢记这一点，因为我们将继续扩展我们的性能词汇表来更好地理解为什么会出现这种现象。在实践中，

吞吐率经常定义为软件能支持的每秒最大交易数。这里的交易代表你的业务领域的一个任务单元（比如订单处理或股票交易）。



思考一下你最近处理过的性能问题，怎么定义该问题？是时延问题还是吞吐率问题？你的解决方案是否在提高吞吐率的同时降低了时延性能？

瓶颈

瓶颈是指系统里最慢的部分。显然，任何系统，包括已经调优的系统，都有瓶颈存在，因为总有某个处理步骤的测量值是整个系统最慢的。注意时延瓶颈不一定和吞吐量瓶颈相同。也就是说，同时可以存在多种不同类型的瓶颈。这也例证了为什么先清楚自己是在对付吞吐量问题还是低时延问题是非常重要的。通过识别瓶颈的过程使你能聚焦在解决关键性能难点。

根据我的个人经验，我们已经见过很多由于忽视了操作环境检查清单而浪费大量时间的案例。有一次，在对付一个广告行业的高吞吐量**实时拍卖**（RTB）平台时，我们跟踪一个吞吐量问题几天都未成功。在启动 RTB 平台后，我们开始优化其吞吐量指标，因为在这个行业里吞吐量是竞争优势。我们的业务团队规定要把指标从 40000 请求/秒提升到 60000 请求/秒作为里程碑。这个问题非常令人头疼，因为这个系统看上去并没有耗尽系统资源。CPU 利用率远低于 100%，而之前尝试提升堆内存空间并没有带来什么提升。

最后灵光闪现来自于我们发现系统是部署在 AWS，默认网络配置是 1Gb 以太网。系统处理的请求大约是 2KB/请求。我们做了下初等算数来计算理论最大吞吐量。1Gb 大约相当于 125000KB，125000KB 除以 2KB 每请求等于理论最大吞吐量 62500RPS。我们用 iPerf 工具做了一次网络测试验证了我们的计算。毫无疑问，我们已经达到了网络极限！

性能总结

我们已经正确定义了几个关于性能的主要概念，即时延和吞吐率，但我们还缺乏明确的量化指标。继续以高速公路上行驶的汽车为例，我们希望找到一种方法来回答以下问题，“从 A 点开到 B 点预期耗时多长？”要回答这个问题，第一步是通过测试多个行驶案例来搜集经验数据。

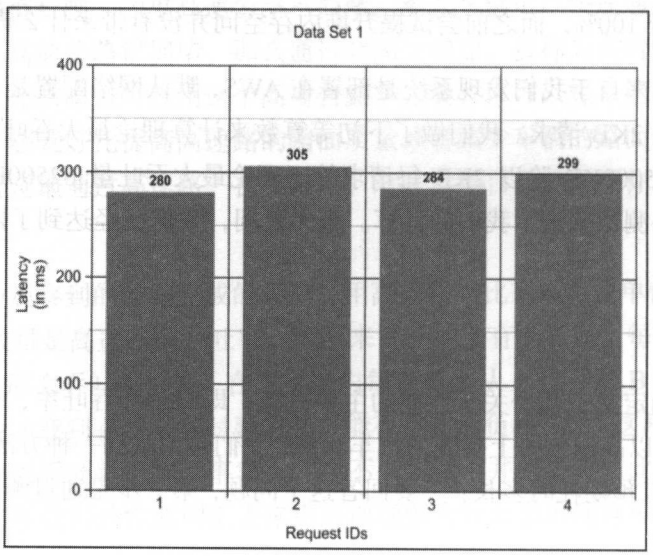
下表收录了我们的观测。我们还需要一种方法来解释和总结测量数据才能回答我们的问题：

观　　测	旅行时间（分钟）
Trip 1	28
Trip 2	37
Trip 3	17
Trip 4	38
Trip 5	18

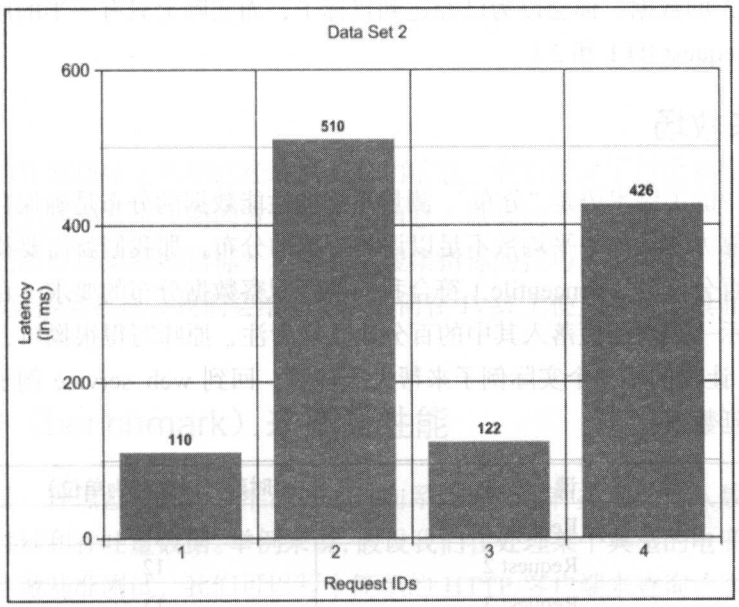
平均数的问题

一个常见错误是依靠平均数来评测系统的性能。数学平均数非常容易计算，把所有搜集到的数据之和除以数据条目的个数即可。根据上表中的样例数据，可以计算出平均行驶时间差不多 27 分钟。通过这个简单例子很容易看出为什么平均数是一个很差的数据。在 5 份观察数据里只有 Trip 1 接近我们的平均数，而其他的都跟平均数差很多。问题的核心在于平均数是一个很失真的统计指标。当我们把一系列数据转化为平均值时信息丢失了，因为不可能通过一个平均数还原原始数据的全部特性。

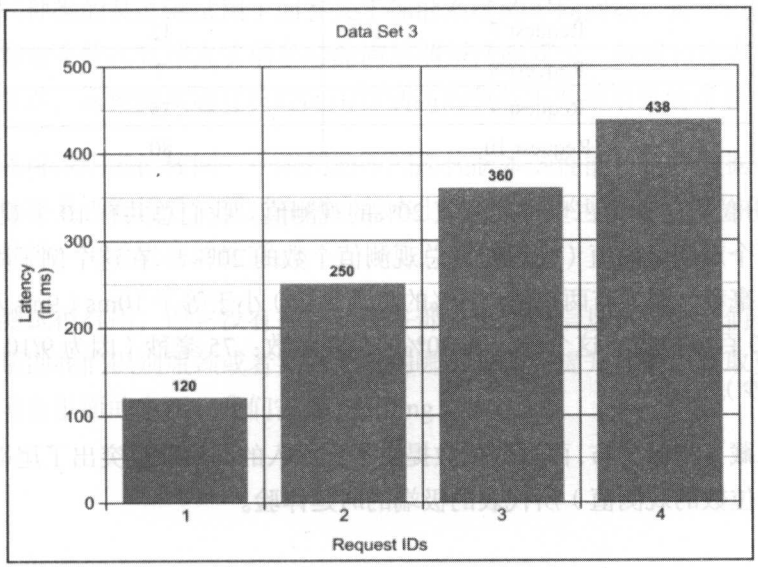
为了展示平均数是如何丢失信息的，考虑以下测量 web service 处理请求时延的所得到的数据集。



第一个数据集包含 4 个请求，每个耗时在 280 毫秒到 305 毫秒之间完成。和下图所示的数据集相比：



第二个数据集显示了一组不太稳定的时延数据。你更愿意部署第一个还是第二个服务到生产环境？为了展示更多不同情况，我们来看第 3 个数据集：



尽管以上数据集的数据分布差异巨大，其平均值是一样的，都是 292 毫秒！假设你在维护数据集 1 所代表的服务，你的目标是保证 75%的用户的响应时间小于 300 毫秒。如果你计算数据集 3 的数据，你会以为已经达到目标了，而实际上只有一半的用户满足你的快速响应要求（request ID 1 和 2）。

百分位数来救场

之前讨论中的关键术语是“分布”。测量系统的性能数据的分布是确保自己真实理解系统行为的最可靠方式。如果平均数不足以评测指标的分布，那我们就需要找个别的工具。在统计学上，百分位数（pencentile）符合我们解释观察数据分布的要求。百分位数是一个统计指标，指示一组观测值落入其中的百分比（译者注，原味写得很拗口，建议大家查一下百度百科）。让我们用一个实际例子来帮大家理解。回到 web service 例子，假设我们观测到以下的时延数据：

请　　求	时延（以毫秒为单位）
Request 1	10
Request 2	12
Request 3	13
Request 4	13
Request 5	9
Request 6	27
Request 7	12
Request 8	7
Request 9	75
Request 10	80

第 20 百分位数定义为占全部观测值 20%的观测值。我们总共有 10 个观测值，所以我们要找代表两个观测值的值（也就是说总观测值个数的 20%）。在这个例子里，第 20 百分位数延迟是 9 毫秒，因为有两个值（20%的总观测数）小于等于 10ms（9ms 和 7ms）。与此相对的是第 90 百分位数。这个值代表 90%的总观测数：75 毫秒（因为 9/10 个观测数小于或等于 75 毫秒）。

平均数隐藏了指标分布，而百分位数提供了更深入的洞察而且突出了尾端的观测值（接近第 100 百分位数的观测值）所代表的极端的时延体验。

如果你还记得在本节开始时，我们想要回答问题，“从 A 点开到 B 点预期耗时多长？”在花了一段时间探索可用工具后，我们注意到初始的问题并不是我们真正关心的问题。更实用的提法是，“90%的车花多长时间从 A 点开到 B 点？”

指标搜集

现在我们的性能指标工具箱已经装满了有用信息。我们定义了讨论和探索性能所用的标准词汇表。我们还同意用实用的方式来汇总信息。旅程的下一步是回答这个问题：为了汇总信息，我们需要搜集哪些指标？本节介绍搜集指标的技巧。下一章，我们会深入聚焦在从 Scala 代码里搜集数据。我们会给你展示如何在 JVM 上使用各种工具和库并且更好地理解你的程序。

用基准数据（benchmark）来衡量性能

基准数据是一种黑盒指标。基准测试通过向系统提交各种类型的输入数据然后测量系统输出来获取时延和吞吐量数据。举例来说，假设我们在处理某个典型的电商购物车应用。为了对这个应用做基准测试，我们可以写个简单的 HTTP 客户端来查询该服务然后记录完成请求的时间。这个客户端可以用来发送不断增加的每秒请求数然后输出一个记录响应时间的总结。

有多种不同种类的基准测试用于回答关于你的系统的不同问题。你可以重放历史生产数据以便确定你的系统在处理真实请求时能满足期望性能要求。负载和压力基准测试识别你的应用的崩溃点，这种测试通过长时间用特别高的负载来考验系统的可靠性。

基准测试还可以用来比较同一个应用的不同迭代版本的性能，这种数据可以用于性能回归或者确定性能是否有提升。通过在两个版本上运行相同的基准测试，你可以实际验证你最近的修改是否产出了更高的性能。

尽管基准测试很有用，但它没有提供软件各部分性能的信息；所以它们是黑盒测试。基准测试不能帮助我们识别瓶颈或者判断系统的哪部分需要提升性能以获取更高的整体性能。为了看看黑盒里面的东西，我们需要 profiling。

通过 Profiling 来定位瓶颈

与基准测试相反，profiling 用于分析应用的内部特征。Profiler 通过捕捉应用各个部分的运行时信息和资源消耗信息使我们可以做白盒测试来识别瓶颈。通过运行时检查你的应用，profiler 可以提供代码行为的极其详细的信息，如下：

- CPU 用在哪里
- 内存用在哪里，对象在哪里初始化，在哪里释放（如果没释放说明存在内存泄露！）
- IO 操作在哪里进行
- 哪个线程在运行、阻塞或空闲

大部分 profiler 通过在编译期或运行期在被观测代码里插入指令来注入计数器和 profiling 组件。这种指令存在运行时开销，会导致系统吞吐率和时延变差。因此，不应该用 profiler 来衡量生产系统的吞吐率和时延（提醒一下，那是基准测试的使用场景）。

一般来说，在决定做任何性能驱动的改进前都应该做 profile。你应该先确定你计划要修改的那部分代码确实是系统的瓶颈。

结合基准测试和 profiling

Profiler 和基准测试的目的不同，解决不同的问题。好的性能提升流程应该把这两种技巧都用上，发挥它们各自的长处来优化代码改进流程。在实践中，一般采取以下步骤：

1. 对当前代码版本跑基准测试来建立性能基线。
2. 用 profiler 来分析系统内部行为并定位出瓶颈。
3. 改进造成瓶颈的部分。
4. 用第一步的相同基准测试来测新版的代码。
5. 用新的基准测试结果和基线比较来确定修改的有效性。

记住有一点很重要，所有的基准测试和 profiling 都应该在相同的环境下跑。参照资源清单来确保所有的测试环境稳定，资源上的任何变化都使测试结果无效。就像科学实验一样，你必须小心地每次只动实现的一个因素。



基准测试和 profiling 在你的开发流程中扮演什么角色？你是否每次在改进代码前做 profiling?你的工作完成定义（definition of done）是否包含基准测试？你是否能在非常接近生产环境的条件下运行你的基准测试和 profiling?

案例分析

我们会在整本书里提供代码例子来演示当前主题。为什么让我们之前介绍的技术在你的专业工作中尽可能地能实际应用，我们会用一个假想的金融交易公司来组织例子，公司名姑且叫作 MV 交易公司。公司名字起源两位作者名字首字母的组合。碰巧这和 Linux 文件操作命令 mv 一致，代表这个公司正在前进中！一年前公司成立至今，MV 交易公司为不多的客户成功地运作了股票交易策略。为了支撑各个相关业务领域，软件基础设施也在一年中快速地建立了起来。MV 交易公司构建了软件来支持多个交易所的实时交易（买入和卖出），同时构建了历史交易分析系统来生成更好的交易算法。如果你没有金融领域知识，不要担心。对于给出的每个例子，我们也会描述关键的业务概念。

工具链

我们建议大家提前安装好需要的软件，方便之后跟着例子操作。因为在工具自己的网站上都提供了详细的安装指南，所以本书就不赘述了。在后面的章节中用到了以下软件：

- Oracle JDK8 以上
- sbt 0.13 以上



代码包下载步骤位于本书的前言里。代码包也同时在 Github 上提供：https://github.com/PacktPublishing/Scala-High-Performance-Programming。我们还提供了其他书的代码包：https://github.com/PacktPublishing/。

小结

在本章里，我们聚焦在用什么术语来讨论性能。我们建立了讨论性能的词汇表，决定了使用百分位来作为性能总结的指标，也建立了对性能评测的直观感受。我们介绍了案例研究，然后安装了需要的软件以便运行代码示例和随书提供的代码。在下一章里，我们会来看一看度量 JVM 性能的可用工具，然后分析我们的 Scala 应用的性能。

2

在 JVM 上度量性能

在前一章里，我们介绍和定义了性能相关的重要概念。虽然那些概念非常重要，但是稍微偏学术了一点，可能你已经等不及要实际着手操练新学到的知识了。恰好本章就是带你实操的！我们将近距离研究一个实际场景，然后深入其代码，分析和度量该应用的性能属性。这个实操章节专注于 MV 交易公司最成功的产品：指令簿（order book）。这是软件的一个关键部分，开发的初衷就是要保证高吞吐量同时维持低时延。在本章中我们将讨论如下主题：

- 对应用的时延和吞吐量做基准测试。
- 用飞行记录器（Flight Recorder）对系统做 profiling。
- 用 JMH 对代码做微基准测试。

金融领域一瞥

本月是 MV 交易公司一周年。在过去的一年里，公司通过新奇的交易策略给客户带来了丰厚的回报。要让这种策略取得最佳效果就需要在获取最新价格信息后的几毫秒内给交易所下交易指令。为了支持低时延交易，MV 交易公司的工程师团队直接把系统集成到了交易所系统里。这包括把软件系统和交易所的系统协同部署在同数据中心以及开发相应的交易系统。

交易系统的一个核心组件，称为指令簿，保持着交易所的状态（state of the exchange），其目的是跟踪有多少买家和卖家在积极操作某个股票，以及双方愿意成交的价格。一个交易员（trader），比如 MV 交易公司，提交订单来买入或卖出某个股票，交易所系统决定交

易什么时候达成，然后将交易信息通知买卖双方。交易所状态由交易所以及 MV 交易公司管理，这个状态是非常有意思的，因为并非所有到达交易所的指令都会被执行。相反，指令可能会直到交易时间（6 个小时）结束都保持在开放（open）或等待（pending）状态。这第一个版本的指令簿系统允许交易员下达一种叫作限制指令（limit order）的指令。限制指令包含一个关于最小成交价格的约束条件。对买家来说，这个限制条件代表交易员所愿意付出的最高价格；对卖家来说，则代表卖家愿意卖出股票的最低价格。指令簿支持的另一个操作是取消未成交的限制指令，该操作会移除指定的指令。为了总结可能的订单状态，我们用下表列出了交易行为所有可能的结果。

交易行为	结　果
限制指令，其价格劣于市场上提供的最好价格	该指令保留在指令簿里，也就是说该指令保持等待状态，直到某个对手方给出的指令符合了成交条件，或者该指令被取消
限制指令，其价格好于等于市场上提供的最好价格	该指令成交（cross the book）。成交是金融行话，是指由于价格符合一个或多个对手盘给出的价格条件，所以交易达成。每一笔交易包含两个执行线（execution），一方一个
取消未成交的指令	未成交指令被移除
取消一个已经成交的指令或者不存在的指令	取消请求被拒绝

假设作为一个刚刚加入 MV 交易公司工程师团队的工程师，你现在负责维护和改进指令簿。今天是你入职首日，计划花一个上午的时间安静地概览一下代码，熟悉一下应用。

从库里检出源代码后，先从领域模型开始看：

```
case class Price(value: BigDecimal)
case class OrderId(value: Long)

sealed trait LimitOrder {
  def id: OrderId
  def price: Price
}

case class BuyLimitOrder(id: OrderId, price: Price)
  extends LimitOrder
case class SellLimitOrder(id: OrderId, price: Price)
  extends LimitOrder
```

```
case class Execution(orderId: OrderId, price: Price)
```

上面的 class 和 trait 定义代表业务概念。我们尤其注意到有两种指令（BuyLimitOrder 和 SellLimitOrder），指令由唯一 ID 标识，包含一个价格，默认用美元单位。



你可能会疑惑为什么我们要为 Price 和 OrderId 创建专门的类定义，它们各自只包装了一个属性（price 里面的 BigDecimal 和 orderId 里面的 Long）。我们应该可以直接用基础类型，不是吗？原因在于 BigDecimal 可以表示各种东西，不仅限于价格，还可以是税率或者经纬度等。定义一个专用类型，取名叫 Price，给 BigDecimal 附加了上下文信息，并且确保编译器能帮我们检查到潜在的错误。我们认为为业务概念显式定义类型是一个好的技术实践。这种技巧也是领域驱动设计里的最佳实践之一，我们在本书中很多地方都使用了这种技巧。想学习更多软件开放方面的技巧，我们推荐 Eric Evans 写的这本非常优秀的书《领域驱动设计——软件核心复杂性应对之道》（http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215）

OrderBook 模块利用上面的领域模型来定义可用的指令和指令可能产出的结果事件：

```
object OrderBook {
  // all the commands that can be handled by the OrderBook module
  object Commands {
    sealed trait Command
    case class AddLimitOrder(o: LimitOrder) extends Command
    case class CancelOrder(id: OrderId) extends Command
  }

  // events are the results of processing a command
  object Events {
    sealed trait Event
    case class OrderExecuted(buy: Execution, sell: Execution)
      extends Event
    case object LimitOrderAdded extends Event
```

```
    case object OrderCancelRejected extends Event
    case object OrderCanceled extends Event
  }

  // the entry point of the module - the current book and
  // the command to process are passed as parameters,
  // the new state of the book and the event describing the
  // result of processing the command are returned
  def handle(book: OrderBook, command: Command): (OrderBook, Event) = //
omitted for brevity
}
```

假设你正打算详细读一读 handle 函数的实现，这时候突然注意到你的技术组长 Alice 发来的即时通信："所有人立刻到会议室，我们有个线上故障！"



有金融背景的读者可能会注意到我们描述的交易行为只是正式金融交易的一个子集。比如交易量就显然缺失了。在我们的例子里，假定每个指令代表对相同数量的股票（比如 100 份）的购买意向。有经验的读者会注意到交易量会使指令簿状态管理进一步复杂化，比如可能引入部分成交的概念。我们有意地简化了这个例子，使它既具有一定的真实性又不至于成为不了解这个领域的读者的理解障碍。

意外的市场波动毁掉了利润

Alice 和交易组长，Dave，一起组织会议对生产故障做了概述。通过会议你对问题增加了很多理解。你了解到，当前市场波动剧烈，价格的快速震荡本来是增加了产生有利可图的交易的机会。但对于 MV 交易公司来说，不幸的是最近几周的剧烈波动同时带来了超出预期的指令数量。为了应对快速的价格变化，市场上充斥着交易员们发出的大量的限制指令和取消指令。MV 交易公司的指令簿已经经过负载测试，能够处理最大每秒 15 000 笔指令（OPS——order per second），且 99%的时延小于 10 毫秒。当前交易市场的 OPS 已经持续超过 45 000，导致指令时延的增加。生产系统部署的指令簿版本目前的第 99 百分位时延已经达到 80 毫秒，而最长时延已经达到 200 毫秒了。在股票交易行业，迟缓的反应很快就

能把原本有利可图的交易变成大笔的损失，这正是当前 MV 交易公司正在发生的情况。通常在一定程度波动的情况下，MV 交易公司能够产生不错的回报，而最近几周则发生了惊人的亏损。我们的目标是应用在“第 1 章 高性能之路”里学到的知识减少性能损失。

MV 交易公司的交易员们在期盼快速的性能提升以便从当前的市场波动中获利。交易员们相信市场波动在平息前还将持续一周。一旦波动消失，获利的机会也就消失了。因此工程师团队承受了巨大的压力，必须要快速地把第 99 百分位数时延降低到 40 毫秒以内，这样应该可以止损并且还能获得少量的收益。一旦波动平息，可以有时间采取更深入和广泛的性能提升措施。时钟正在滴答作响，我们必须找出办法提高性能来止损。

重现故障

你没想到入职的第一天会是这个，不过这挑战多刺激啊！我们先从重现故障开始对性能问题做调研。如我们在第 1 章里所说的，对应用建立一个基线是极其重要的事情。基线用于评估我们采取的性能提升措施是否有效。我们创建了两个简单的基准测试来重现生产上观察到的负载压力，度量系统的吞吐量和时延。



等一下，我还没看完 OrderBook 是怎么实现的！你是对的！你对于模块的实现确实还毫无头绪。但是生产系统挂了，我们必须要快速反应！更重要的是，这正是基准测试的特点。我们在第 1 章提过，基准测试是把应用当作黑盒。你已经看过模块的接口，而这已经够让你写基准测试了。

吞吐量基准测试

我们的第一个基准测试测量应用的吞吐量。运维团队给我们提供了生产系统以前记录下来的历史数据。这些数据包含了几十万条曾经在指令簿模块处理过的实际指令。我们用对测试环境重放（replay）这些样本数据来获得对系统行为的准确认识。



回忆一下在第 1 章里讲过，在相同的环境下运行所有的基准测试是非常重要的事，这样才有比较的意义。为了确保跨测试的一致性，我们构建了一个指令生成器，能够输出一组静态的指令集。建议你复查一下。

下面是吞吐量基准测试的代码，你可以在章节的子项目代码里找到：

```
object ThroughputBenchmark {
  def main(args: Array[String]): Unit = {
    val commandSample = DataCodec.read(new File(args(0)))
    val commandCount = args(1).toInt
    jvmWarmUp(commandSample)
    val commands = generateCount(commandSample, commandCount)
    val start = System.currentTimeMillis()
    commands.foldLeft(OrderBook.empty)(OrderBook.handle(_, _)._1)
    val end = System.currentTimeMillis()
    val delayInSeconds = (end - start) / 1000.0
    println {
      s"""
         |Processed ${commands.size} commands
         |in $delayInSeconds seconds
         |Throughput: ${commands.size / delayInSeconds} operations/sec"""
        .stripMargin
    }
  }
}
```

代码很直观，所以我们搜查一遍。首先，我们读入输入参数，第一个是参数指向历史数据的文件路径，第二个是我们打算跑的指令数量。注意在当前的实现下，如果我们想要跑的指令数量超过静态文件里的指令数量，那程序只会把静态文件里的指令全跑一遍而已。然后我们先执行最多 10 万条指令而不记录吞吐量数据，从而达到预热 JVM 的目的。预热的目的是让 JVM 有机会分析执行的代码，从中找到热点，然后用**即时编译器**（just-in-time——JIT 编译器）来优化它。



JIT编译器在应用运行之后才运行。它会把字节码（也就是之前由javac编译器编译出来的结果）即时编译成更优化的形式，一般是当前操作系统的本地代码。JIT能够根据运行时使用的情况来优化代码。这是传统编译器做不了的事情，因为传统编译器是在代码执行前运行的。

接下来的代码是我们实际记录吞吐量的代码。我们记录开始时间戳，对空指令簿执行所有的指令，然后记录结束时间戳。通过把指令数量除以花费的时间，我们很容易就得到了每秒吞吐量。因为吞吐量的计算单位是秒，所以起止时间记录到毫秒级就足够了。最后把感兴趣的结果打印出来。我们可以用命令行参数指定跑25万个指令的基准测试，像这样：

```
sbt 'project chapter2' 'set javaOptions := Seq("-Xmx1G")'
'runMainhighperfscala.benchmarks.ThroughputBenchmarksrc/main/resources/historical_data 250000'
```

对一组不同数量的指令集跑基准测试得到如下结果：

指令计数	处理时间（秒）	吞吐量（每秒操作数）
250,000	2.2	112,309
500,000	6.1	81,886
750,000	12.83	58,456
1,000,000	22.56	44,328

我们可以看到，随着指令数量增长，我们的吞吐量下降。一种解释是由于指令簿本身的大小随着接到更多指令而增长，其性能也就逐渐下降了。在本节里，我们测量了应用的吞吐量。 在下一节里，我们专注于测量程序的时延。



我们当前写的和本章后面将要写的基准测试是过于简单的。 它在同一个虚拟机里运行测试和地址簿。更实际的例子应该单独启动指令簿，监听TCP连接，接受发来的FIX消息（FIX是金融行业最为广泛采用的协议）来创建或取消指令。而我们的基准测试应该模仿成一个客户端来模拟对地址簿的生产负载。为了简化的目的以及让我们能够把注意力放在更有意思的主题上，我们决定搁置这个顾虑。

时延基准测试

回忆我们在第 1 章里所说的，时延是指一个操作完成所花费的时间，操作的定义由你的业务领域决定。在我们的案例里，我们把操作定义为从指令簿收到新的指令到相关结果事件生成这样的一个过程。

第一个时延基准测试

下面的代码是我们的时延基准测试的第 1 版：

```
object FirstLatencyBenchmark {
  def main(args: Array[String]): Unit = {
    val commandSample = DataCodec.read(new File(args(0)))
    val (commandsPerSecond, iterations) = (args(1).toInt, args(2).toInt)
    val totalCommandCount = commandsPerSecond * iterations

    jvmWarmUp(commandSample)

    @tailrec
    def sendCommands(
      xs: List[(List[Command], Int)],
      ob: OrderBook,
      testStart: Long,
      histogram: HdrHistogramReservoir): (OrderBook, HdrHistogramReservoir)
      =
      xs match {
        case head :: tail =>
          val (batch, offsetInSeconds) = head
          val shouldStart = testStart + (1000 * offsetInSeconds)

          while (shouldStart > System.currentTimeMillis()) {
            // keep the thread busy while waiting for the next batch to be
            sent
          }

          val updatedBook = batch.foldLeft(ob) {
            case (accBook, c) =>
```

```
            val operationStart = System.currentTimeMillis()
            val newBook = OrderBook.handle(accBook, c)._1
            val operationEnd = System.currentTimeMillis()
            // record latency
            histogram.update(operationEnd - operationStart)
            newBook

          }

          sendCommands(tail, updatedBook, testStart, histogram)
        case Nil => (ob, histogram)
        }

      val (_, histogram) = sendCommands(
        // Organizes commands per 1 second batches
        generateCount(commandSample, totalCommandCount)
          .grouped(commandsPerSecond).zipWithIndex
          .toList,
        OrderBook.empty,
        System.currentTimeMillis(),
        new HdrHistogramReservoir())
      printSnapshot(histogram.getSnapshot)
    }
  }
```

代码的开头部分和吞吐量测试差不多，我们用 HdrHistogram 来记录每个操作的时延。尾递归方法 sendCommands 是最有意思的部分（我们会在后面的章节详细讨论尾递归）。我们的指令按照尺寸和 commandsPerSecond 分组，也就是说我们每秒钟发送一批。我们记录下指令发送前的时间（operationStart）和接到响应后的时间（operationEnd）。这些时间戳用来更新**直方图**（histogram）。



HdrHistogram 是直方图的一种高效实现。专门设计用于实验和性能敏感的应用。它在空间和时间上都保持固定开销。它不涉及到内存分配操作，其内存占用是常量。要了解更多关于 HdrHistogram 的知识，请访问 http://hdrhistogram.org/。

最后，当所有批次都处理完后，我们给直方图的状态做个快照然后输出感兴趣的指标。来运行一下看看：

```
sbt 'project chapter2' 'set javaOptions := Seq("-Xmx1G")' 'runMain
highperfscala.benchmarks.FirstLatencyBenchmark
src/main/resources/historical_data 45000 10'
... // removed for brevity
[info] Processed 450000 commands
[info] 99p latency: 1.0 ms
[info] 99.9p latency: 1.0 ms
[info] Maximum latency: 24 ms
```

我们试着给系统按照每秒 45000 笔操作的速率发送 10 秒钟，然后可以看到大约 99.9% 的操作时延小于 1 毫秒。这结果不可思议！然而它是完全错误的。因为太着急写时延基准测试程序，我们忽视了一个经常被忽略的问题：协同遗漏问题（coordinated ommission problem）。

协同遗漏问题

我们的基准测试失败，是因为我们在测量处理一个指令需要的时间时，没有考虑到指令等待被处理的时间。当**前一个**指令处理的时间比预期时间长的时候，问题就出现了。拿前面的例子来说：我们每秒跑 45000 个指令，也就是说，每毫秒 45 条指令。如果处理前 45 条指令的时间超过 1 毫秒会怎样？后 45 条指令就不得不等待轮到自己被处理。然而我们当前的基准测试忽略了等待时间。用一个提供网页服务的 web 应用来打比方：典型的基准测试会通过计算——请求被 web 服务器接到开始处理——到发回响应结果之间所花费的时间，来记录请求时延。而更好的基准测试程序会测量客户端发出请求的时间到它实际接到响应结果的时延。要了解更多关于协同遗漏问题的知识，请参考以下讨论：`https://groups.google.com/forum/#!msg/mechanical-sympathy/icNZJejUHfE /BfDekfBEs_sJ`。[1]

为了修复这个问题，我们需要把 `operationStart` 记录为，一个批次的指令已经开始被处理的时间，而不是已经开始处理的时间，不论系统是否延迟。

1 其实就是说上面的程序只记录了 `OrderBook.handle` 处理一条指令的时间，而没有记录这条指令在处理前在批次里等待的时间。——译者注

时延基准测试程序第 2 版

在第 2 次尝试中，我们要确保在指令应当被处理的时候开始时钟，而不是当指令马上要被处理前。

基准测试代码大部分不用修改，除了记录时延的部分，现在改用 shouldStart 而不是 operationStart：

```
histogram.update(operationEnd - shouldStart)
```

在这次修改后，新基准测试的输出如下：

```
sbt 'project chapter2' 'set javaOptions := Seq("-Xmx1G")' 'runMain
highperfscala.benchmarks.SecondLatencyBenchmark
src/main/resources/historical_data 45000 10'
... // removed for brevity
[info] Processed 450000 commands
[info] 99p latency: 743.0 ms
[info] 99.9p latency: 855.0 ms
[info] Maximum latency: 899 ms
```

结果和第一次相差巨大。实际上，新版本也有瑕疵。因为它假设在同一秒发出的所有请求都应该在该秒的一开始就开始处理。虽然技术上不是不可能，但更大的可能是这些指令在这一秒里分时地发出，有些在第 1 毫秒发出，有些在第 2 毫秒发出，以此类推。这个版本的基准测试很可能过高估计了时延，因为它计时器启动得过早了。

时延基准测试程序最终版

我们尝试修复最后一个问题，最终拿出一个可靠的基准测试。这次我们要尝试解决一秒内的指令分布的问题。解决这个问题的最好办法是使用真实生产数据。如果用于基准测试的生产数据里面包含时间戳（生产系统实际接收到指令的时间戳），我们就可以复制生产系统上所遇到的指令分布情况。

不幸的事，当前版本的指令簿应用没有记录时间戳。我们可以想想别的办法。一个办法是在一秒内随机发送指令。另一个办法是平均分布指令（也就是说每毫秒发送相同数量的指令）。

我们决定用第 2 种思路修改基准测试程序。为了达到目的，我们修改事件生成的方法。

因为我们的新策略随时间分布指令而不是批次发送指令，对一个实例来说，新的指令列表返回类型从 List[(List[Command]),Int)]修改为 List[(Comand,Int)]。生成指令表的逻辑考虑到我们的新的策略所以修改成这样：

```
generateCount(sampleCommands, totalCommandCount)
  .grouped(cps.value)
  .toList.zipWithIndex
  .flatMap {
    case (secondBatch, sBatchIndex) =>
      val batchOffsetInMs = sBatchIndex * 1000
      val commandIntervalInMs = 1000.0 / cps.value
      secondBatch.zipWithIndex.map {
        case (command, commandIndex) =>
          val commandOffsetInMs =
            Math.floor(commandIntervalInMs * commandIndex).toInt
          (command, batchOffsetInMs + commandOffsetInMs)
      }
  }
```

指令生成的代码变得稍微复杂了一点。我们现在为每一条指令计算一个偏移量，考虑到了每条指令之间的时间间隔毫秒数。最后的基准测试结果如下：

```
sbt 'project chapter2' 'set javaOptions := Seq("-Xmx1G")' 'runMain
highperfscala.benchmarks.FinalLatencyBenchmark
src/main/resources/historical_data 45000 10'
[info] Processed 450000 commands
[info] 99p latency: 92.0 ms
[info] 99.9p latency: 137.0 ms
[info] Maximum latency: 145 ms
```

终于为我们的系统建立了不错的实验基准测试，而且看上去这个结果非常接近我们生产系统的实际情况。



我非常希望这个例子能够让你想到你自己生产系统的现状以及你想要做的基准测试类型。你从本节应该学习到协同遗漏问题的重要性。你能想到的测试你自己的系统的时延基准的最好方法是什么？如果你已经有写好的基准测试，它有没有考虑到协同遗漏问题？

定位瓶颈

现在我们已经有了能够一致地重现生产性能的基准测试，有信心准确地测量我们的任何代码修改的效果。基准测试把指令簿视为黑盒，也就是说它不能告诉我们，发现是指令簿的哪部分代码造成了性能问题。如果你原本很熟悉代码，也许能从中获取一些启发，让你比较好地猜测哪些子组件有问题的可能性更大。由于这是你入职第一天，你没有之前的经验可以依靠。与其乱猜，不如对指令簿做个 profile 来获取对这个黑盒的不同层面的更深入的理解。

JDK 自带了一个优秀的 profiler，叫作 Flight Recorder。FlightRecorder 可以在非生产环境免费试用。请参考 Oracle 的协议 `http://docs.oracle.com/javacomponents/jmc-5-5/jfr-runtime-guide/about.h tm`来了解关于商业试用的规定。有一个强大的 profiler 的存在是另一个实在的理由——选择 Scala 来作为生产质量函数式编程的语言。FlightRecorder 用 JVM 内部钩子（hooks）来记录 JVM 运行时发出的事件。FlightRecorder 捕捉的事件包括内存分配、线程状态修改、IO 活动和 CPU 活动。想学习更多 Flight Recorder 内部实现的知识请参考 Oracle 官方文档：`http://docs.oracle.com/javacomponents/jmc-5-5/jfr-runtime-guide/about.htm#sthref7`。与其他不能访问 JVM 内部信息的第三方 profiler 不同，Flight Recorder 可以访问超出 JVM 安全点（safe point）的数据。JVM 安全点是指所有的线程都挂起的时点。安全点对于协调 JVM 全局活动，包括全局垃圾回收，是非常必要的。想了解更多关于 JVM 安全点的知识，请参考 Alexey Ragozin 的优秀博客文章 `http://blog.ragozin.info/2012/10/safepoints-in-hotspot-jvm.html`。如果 profiler 只能在安全点来检查系统状态，很可能它会遗漏一些有用数据，因为它只能看到整个全景画面的一部分。

我们先来配置一下 Flight Recorder 试用版以便透视指令簿应用。为了加快开发周期，我们通过 `sbt` 来配置 profiler，让它在运行 `ThrougoutBenchmark` 回放历史数据时执行。用如下参数配置 Flight Recorder：

```
sbt 'project chapter2' 'set javaOptions := Seq("-Xmx1G", "-
XX:+UnlockCommercialFeatures", "-XX:+FlightRecorder", "-
XX:+UnlockDiagnosticVMOptions", "-XX:+DebugNonSafepoints", "-
XX:FlightRecorderOptions=defaultrecording=true,dumponexit=true,dumponexitpa
th=/tmp/order-book.jfr")'
```

为匹配我们的基准测试需要，最大 JVM 堆大小设置后面加上 4 个 JVM 参数。

-XX:+UnlockCommercialFeatures 和 -XX:+FlightRecorder 参数用于为 FlightRecorder 产生 JVM 事件。参考 Flight Recoder 文档，-XX:+UnlockDiagnosticVMOptions 和-XX:+DebugNonSafepoints 用于提高采样质量。这些配置项指示编译器生成元数据，使 Flight Recorder 可以捕捉不在安全点的采样。最后一个参数配置程序一开始执行就启动记录 FlightRecorder 数据，并且在退出时把 profile 结果输出到指定路径。以我们的例子来说，这意味着 profile 会在基准测试启动时开始，在基准测试完成时终止。除了这种配置外，还可以配置 Flight Recorder 推迟启动时间并且记录固定的一段时间后终止，参考如下配置：

```
sbt 'project chapter2' 'set javaOptions := Seq("-Xmx1G", "-
XX:+UnlockCommercialFeatures", "-XX:+FlightRecorder", "-
XX:+UnlockDiagnosticVMOptions", "-XX:+DebugNonSafepoints", "-
XX:StartFlightRecording=delay=10s,duration=60s,name=Recording,filename=/tmp
/order-book.jfr")'
```

上面的配置让 Flight Recorder 在 5 秒后启动（delay 选项），然后记录一分钟（duration 选项）。结果存放在/tmp/order-book.jfr。

现在我们准备好生成 profile 结果了。接着，运行基准测试重放两百万个请求。重放的请求越多，profiler 就有越多的机会捕捉 JVM 事件。如果其他条件都一样，那就选择尽量长时间的 profile。下面的输出显示了基准测试调用的情况和结果输出：

```
sbt 'project chapter2' 'set javaOptions := Seq("-Xmx1G", "-
XX:+UnlockCommercialFeatures", "-XX:+FlightRecorder", "-
XX:+UnlockDiagnosticVMOptions", "-XX:+DebugNonSafepoints", "-
XX:FlightRecorderOptions=defaultrecording=true,dumponexit=true,dumponexitpa
th=/tmp/order-book.jfr")' 'runMain
highperfscala.benchmarks.ThroughputBenchmark
src/main/resources/historical_data 2000000'
  [info]
  [info] Processed 2000000 commands [info] in 93.276 seconds
  [info] Throughput: 21441.742784853555 commands/sec
```

我们用 Java Mission Control（JMC）来查看 profiling 结果，这是一个免费的、内置 GUI 的工具，能够查看 Flight Recorder 结果，还能运行 Flight Recorder profile 会话。JMC 和 Java 执行程序位于相同路径下，也就是说你可以直接敲入命令：

```
jmc
```

GUI 加载完成后，通过菜单 File | Open 来打开 profile 结果。浏览到 profile 结果文件，单击 OK 来加载。在查看结果的同时，我们将构造一个问题清单来帮助我们考虑如何复查 profiler 结果。这些探索问题的目的是让你批判式地分析结果。这些问题确保实验结果确实能够验证我们对问题的假设。在本章末尾，我们会给出一个单独的问题清单方便后续使用。

我是否用预期的资源来做测试？

如果测试环境设置得不正确，那么结果就无意义。因此有必要首先仔细检查环境设置。幸运的是，Flight Recorder 为你捕捉了大部分这类信息。

General 和 System 页上的信息专注于这个问题。在 General 页，单击 JVM 信息来检查关键的 JVM 信息。在这部分里，确认以下信息：

关注的领域	关注原因
JVM 启动时间	快速检查确认测试是否在你期望的时间开始。如果你做了好几次 profile，确保没搞错结果文件是一件重要的琐事。随着你搜集越来越多的信息并且调查更多的假设，这个检查能确保你没有把测试结果搞混
JVM 版本	JVM 的不同版本可能产生不同的结果。请确保你用的 JVM 版本和生产环境一致
JVM 命令行参数和 JVM 标志	通过命令行参数微调 JVM 是很常见的做法。经常会发生每次运行之间修改一点参数的情况，这使得之后回忆起来哪个运行结果对应哪个运行参数变得困难。复查这个信息可以提供有用的上下文
Java 应用参数	类似于前面的顾虑，这个检查的目的是确保你理解了测试的输入

为了补充确认 JVM 配置，查看 Memry 页下面的 GC Configuration 页。这页的信息详细描述了垃圾回收配置，反映出用户提供配置项和 JVM 默认配置项的组合效果。你很可能已经知道，在垃圾回收配置上的小小修改可能导致运行时性能的很大差异。考虑到应用性能对垃圾回收配置的敏感性，你应该仔细复查这页的参数配置。应当考虑的问题包括：

- 如果我配置这个参数，这个配置值会生效吗？
- 如果我不配置这个参数会有什么效果？这个问题经常能帮助你产生后续的 profile 假设。

接下来，我们专注于 System 页。Overview 页列举了非 JVM 资源，以便明确哪些资源被用来产生这次的 profile。继续 General 页提出的问题的中心思想，首要的目标如下：

- 确认记录的信息是否符合你预期的环境资源（比如可用内存是否符合你预期的大小）。
- 查找测试环境和生产环境超出预期的差异（比如，我的本地机器是 3.18.x 内核而生产环境是用的旧版本 3.17.x）。

如果你是用环境变量来配置系统，那还有一个 Environment Variables 页应该要检查。和 Overview 页类似，你在确认你的测试环境整备的符合预期。值得反复提醒的是测试环境的意外差异往往会使测试结果无效。

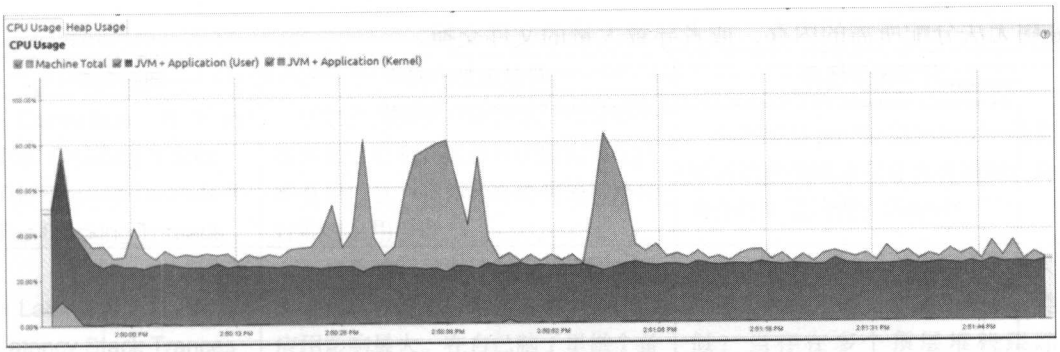
在 profiling 的过程中系统环境是否干净？

一旦你确定了资源环境是正确的，下一步是确定只有被 profile 的应用使用这些资源。这是检查测试结果前的重要诊断步骤，因为这确保了被 profile 的应用真正地被隔离测试。幸运的是，Flight Recorder 罗列了有用的信息来回答这个问题。在 System 页组，Process 页捕捉了在 profiling 期间运行的进程。带着下面的问题检查该列表：

- 当我检查进程列表时，有没有看到不该运行的进程？
- 当我用命令行列来过滤进程并输入 java 时，是否看到预期的 JVM 应用在运行？
- 当我查看进程列表时，有没有看到重复的进程？

接着，检查 General 页组下的 Recording 页。Flight Recorder 提供了创建并行记录的能力。这个 profile 结果会包含并行记录的联合结果。如果意外地在几次运行中有一次发生了并行运行，可能会产生牛头不对马嘴的测试结果比较。

下一个要关注的是 profiling 期间的系统 CPU 利用率。在 General 页组下选择 Overview 页。这个视图显示 CPU 利用率面板，该面板提供了检查测试期间 CPU 使用的能力。在这个页面，你要检查是否存在未预期的 JVM 和机器总 CPU 利用率的分歧。下面的截图展示了一个值得仔细检查的分歧：



预期之外的非JVM CPU使用情况（在CPU利用率面板里被高亮显示）

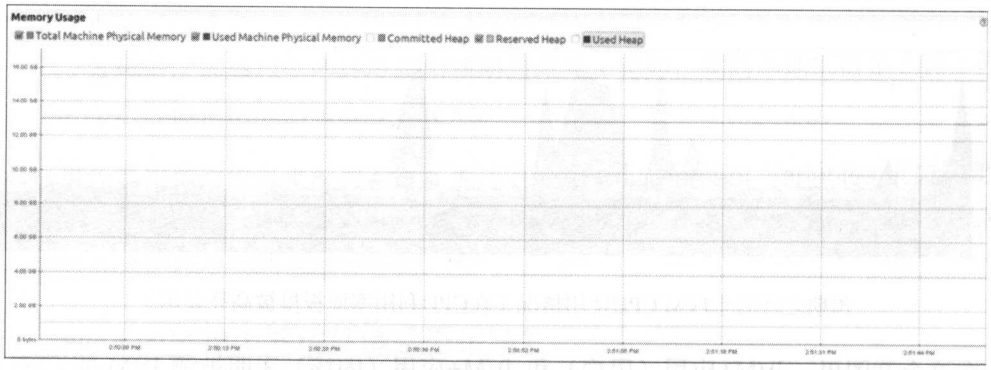
在这个截图里，JVM+应用（用户）和 JVM+应用（内核）表明被测 JVM 的 CPU 利用率，而 Machine Total 表明机器（也就是被测 JVM 加上所有其他进程）的 CPU 利用率。记录的大部分情况下，被测 JVM 的 CPU 利用率占了机器 CPU 利用率的主体。如果你的应用是唯一使用系统资源的进程，那么 JVM+应用（用户）和 Machine Total 之间的小差距恰好代表了理想状态。记录期间中部的分歧反映出有另外一个或多个进程在使用 CPU 资源。这些波峰提示存在异常情况可能会对 profile 结果产生负面影响。值得去研究到底是什么别的进程在使用系统资源以及你的测试结果是否还有效。

现在是介绍 range navigator 的好机会，也就是每个页面上包含小红叉的那个小小的水平控件。range navigator 是一个时间线显示当前页面随时间发生的事件。默认情况下，整个时间线都被选中，profiling 的持续时间在时间线的正中。你可以选中时间线的子集以便放大某个感兴趣的领域。比如说，你可能想放大 CPU 利用率波峰期间的 CPU 利用信息。当选中时间线的子集并且某个数据只在时间线的特定点有效（比如记录开始时）或者数据不表示时间序列（比如 JVM 版本），则该数据被隐藏或替代为 N/A。

最后一个检查点是在 Memory 页组下 Overview 页里的 Memory Usage 面板里检查机器物理内存使用。检查机器物理内存是否是一个有意义的值。如果已经几乎没有可用物理内存而保留堆内存是全部机器物理内存的一个小百分比，你应该暂停以考虑到底有什么其他进程在使用内存。下图演示了这样一个场景：

在这个例子里，保留堆空间是总共 16GB 里面的 2GB，而 13GB 系统内存正在被使用。这隐含说明 11GB 的系统内存被别的进程占用了。除非你本来就期望有别的进程在你的测试环境下运行，否则这种类型的内存矛盾情况就很值得进一步调查。比如说，如果你的应用使用了堆外内存（off-heap memory），这种矛盾情况就可能导致你的测试无效，因为你的

应用无法分配所需的内存，或者导致大量的文件交换。



非JVM进程使用了大部分可用内存（被内存使用面板捕捉到了）

JVM 内部资源执行是否符合预期？

我们的 profiling 检查清单从最宽泛的问题开始，首先验证环境资源是否整备和配置正确。接着我们缩小范围，聚焦于 JVM 配置以便确保测试结果出自正确的配置。现在我们开始检查 JVM 内部细节以继续验证 profile 结果没有被搞废。

几乎所有的生产应用都使用了多线程来更好地利用多 CPU 核，或者把 I/O 敏感的操作和CPU为中心的操作分离。Threads 页组帮助你掌握应用内的工作分离(division of labor)，并给你线索让你考虑哪些部分值得更深入地探寻。下表列出在 Threads 页组里需要关注的部分，并且突出了当你是新手或者不熟悉被测软件，又或者你有几份 profile 结果要比较时应当提出的问题：

关注领域	不熟悉被测软件	熟悉被测软件
Overview 页的 Thread Count	总共存在多少线程？是否和你原本预期的有差别？比如，如果应用是 CPU 密集型但是却有超过核数 10 倍的线程存在，这可能是一个警告信号说明应用微调得很差	在不同的 profile 结果间是否有什么定性的修改？比如线程数多一倍或者少一半可能提示一个配置错误
Hot Threads 页下的 Hot Threads	线程的分布是否平均，还是说有几个线程主导了采样数据？最热的几个线程很可能提示你应该深入理解并且熟练掌握那部分代码	在线程采样计数分布（thread sample count distribution）之间是否有显著改变？如果是，这些改变对你是否有意义

续表

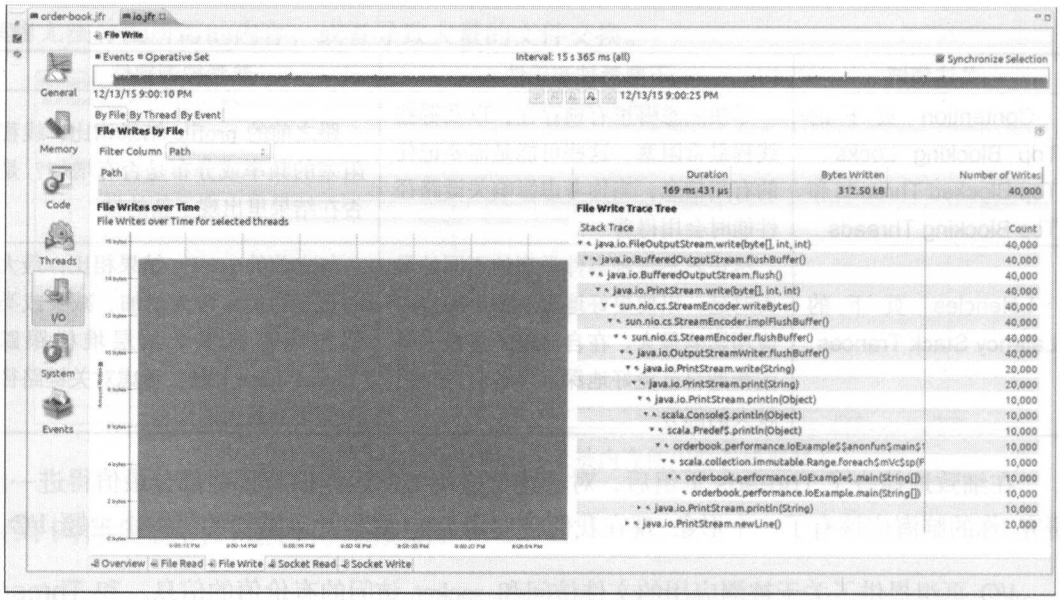
关注领域	不熟悉被测软件	熟悉被测软件
Contention 页下的 Top Blocking Locks、Top Blocked Threads 和 Top Blocking Threads	尽快熟悉哪里有锁存在，以及哪些线程最常阻塞。这些可能是需要记住的有用信息，当你考虑影响关键路径性能时会用得着	跟之前的 profile 结果相比，线程阻塞的频率或分布是否有增加？是否在结果里出现了新的锁
Latencies 页下的 Latency Stack Trances	尽快熟悉每种事件类型的不同的最大时延，以便更好地理解哪个操作对应用影响最大。在自己脑子里做个备注帮助自己更好地深入理解应用延迟较大的部分	与之前的 profile 结果相比，最大时延是否能定性为增加、减少或类似？当存在多个顶层堆栈跟踪（stack trace）时，考虑对关键路径性能影响最大的那些

在细致地检查了 Threads 页组后，对于这个应用如何运作以及哪些部分最值得进一步研究，你的脑海应该有了一个形象。现在我们转向跟JVM线程关系密切的另一个主题：I/O。

I/O 页组提供了关于被测应用的文件访问和 socket 访问的有价值的信息。和 Threads 页组的复查类似，这部分可能给你应用行为意外的或者不理想的提示线索。在深入这部分前，暂停一下来考虑什么时候、什么东西会导致磁盘读写和网络读写。在你复查 Overview 页时，有没有看到你原本以为的和 profile 结果实际显示的行为之间的分歧？如果是，你应该检查这种矛盾为什么存在以及这是否导致测试结果无效。

意外的 I/O 行为的例子比如对标准输出的过度写入。这有可能发生在误打开了 debugging 开关。想象如果这种副作用发生在应用的关键路径上，它会对 profile 结果产生负面影响导致测试无效。在 Flight Recorder 里，写入标准输出被捕捉为空写入路径。下面的截屏显示一个简单的一次性运行的应用反复地高频率地写入标准输出。

为了效果，这个例子有点夸张，所以看上去有一块持续的写入块。通过检查 File Write 页，我们可以看到有多少时间花在写入标准输出，写了多少数据量，以及发生了多少次写入。在大概 15 秒的执行时间里，发生了吓人的 40 000 次写入！文件写入堆栈跟踪提供了非常有价值的信息，让我们可以回溯来定位应用的哪部分要为写操作负责。Flight Recorder 允许你按照线程查看写入。在一个有专用的写线程的生产应用上，你可以很快地隔离出计划外的 I/O 访问。



通过I/O面板组发现预期之外的对标准输出的写入

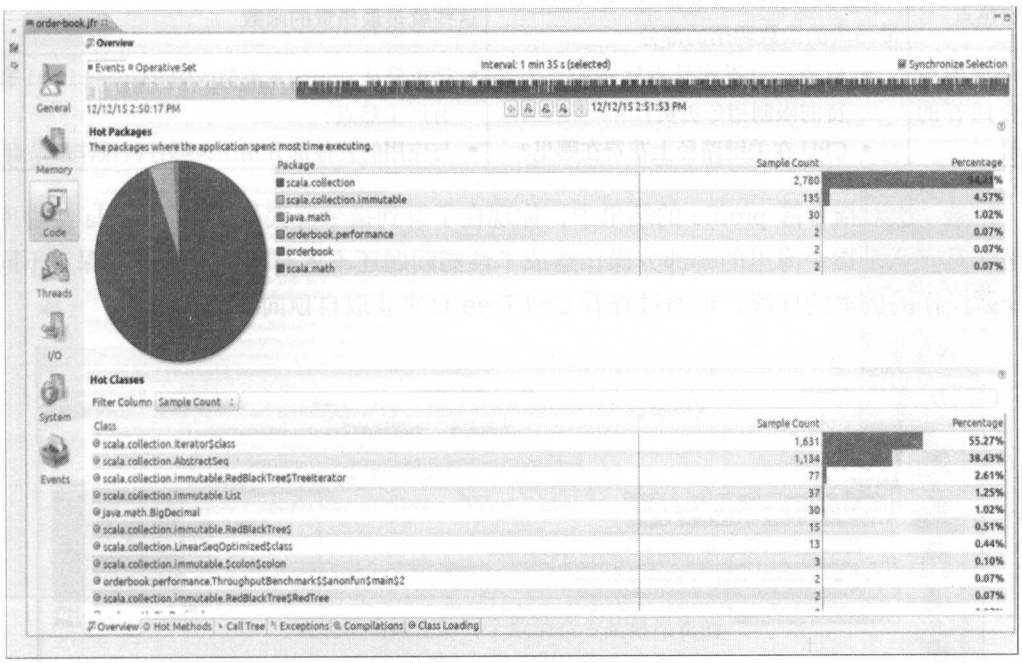
监控 I/O 读写提供了讨论怎么配置 Flight Recorder 记录参数的好机会。`-XX:+FlightRecorderOptions` 接受一个叫作 `settings` 的参数，默认指向 `$JAVA_HOME/jre/lib/jfr/default.jfc`。你可以提供自己的配置文件，也可以修改默认文件。在这个配置文件里，你可以调节 Flight Recorder 记录哪些事件以及捕捉指定事件的频率。比如，`java/file_write` 事件的默认配置是 20ms 阈值。这是一个合理的默认值，但是如果你正在专注于分析文件读写，你可能希望把该值稍微调低。把阈值调低意味着会捕捉更多的事件采样。请谨慎微调，因为并非总是越低越好。更低的阈值意味着更高的记录开销和更多的信息需要排查。

最后一个需要调研的领域是 Code 页组下的 Exception 页。即使你在 profiling 期间细致地监控应用日志，你仍然不一定会把日志保存下来留待历史分析。幸运的是，Flight Recorder 捕捉了异常和错误供复查用。扫描异常和错误，记录总共发生了多少异常和错误，其中哪些发生得最频繁。利用 range navigator 来收缩时间轴以更好地理解异常和错误是否集中在应用启动时、测试的后期，还是说一直在发生。异常和错误发生的时机经常提示根由是否是错误配置或者意外的运行时行为。一如往常，如果你注意到多得吓人的异常或错误，应视这次测试解决为无效，直到你更深入地理解了它们发生的原因。

CPU 瓶颈在哪里？

到现在，我们终于确定了 profiler 结果是有效的，值得进一步研究。现在我们开始 profiling 过程里可能是最有意思的部分：定位 CPU 瓶颈。这个过程很有乐趣，因为 profiler 给出了应用黑盒内部的详细信息。这提供了一个机会，来客观地检验你对应用是如何运行的假设和思维模型。一旦定位到瓶颈，你会感到一丝快慰：现在你精确地知道应该修改哪部分代码来提升应用性能了。

我们从 Code 页组的 Overview 页开始。这个视图可以让你自底向上地对代码的哪部分最昂贵建立一些实际感受。下图展示地址簿一次运行后的 Overview 页：



代码概览面板对昂贵的包和类做了摘要

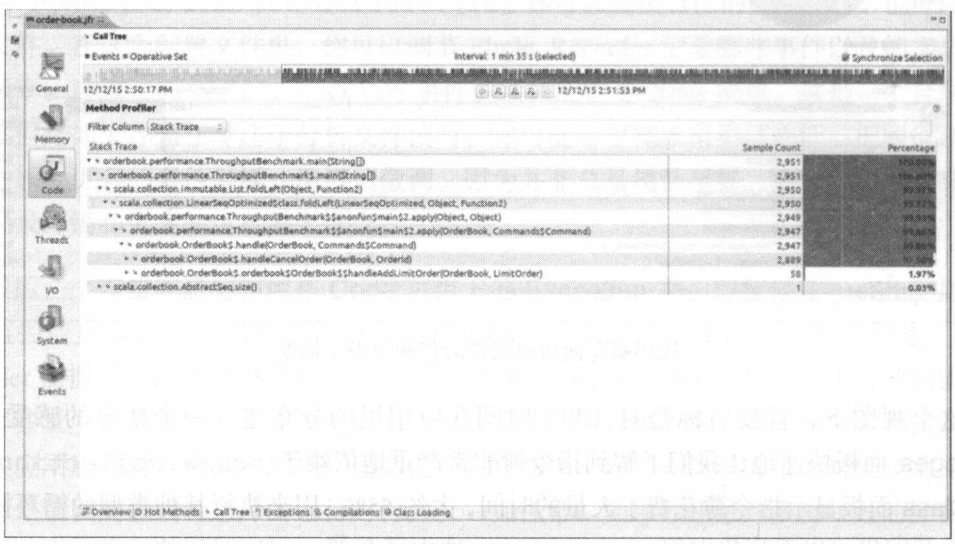
这个视图下，首要目标是对 CPU 时间在应用里的分布建立一个初步的感受。Hot Packages 面板快速地让我们了解到指令簿非常严重地依赖于 `scala.collection` 包。Hot Class 面板显示指令簿花费了大量的时间，大约 55%，用来执行某种类型的循环操作。在这个截图里可以看到我们用 range navigator 选中了全部 profile 时间的一个子集。一个有用的手段是查看 profile 期间的不同子集，判断热点是否一直相同。在这个例子里，我们排除了早期的部分，因为其中包含了待发送的请求的部分。选择这个阶段使我们能够专注于

指令簿操作的部分而不受测试准备期间的干扰。

有一个要点需要提醒注意，百分比列指出的是花费在执行显示的包和类上的应用时间。这意味着这个视图以及 Hot Method 页，是个自底向上的视图而不是自顶向下的视图。在自顶向下视图里，百分比列指出的是花在部分堆栈跟踪（stack trace）上的时间的和。这个视图叫作 Call Tree。这个区分很关键，因为这两种视图帮助我们回答不同的问题。下表从这两个视角探索了几个主题，让我们更好地理解哪种情况下某种视图更有帮助。

主　题	自顶向下（Call Tree）	自底向上（Overview/Hot Method）
视角	从宏观到微观	从微观到宏观
确定热点	这些是代码库里委派给（delegate to）最昂贵的函数调用的部分	这些就是最昂贵的函数
每种视图最适合回答的问题	• 取消指令操作是否比添加一个等待的限制指令更贵？ • CPU 在关键路径上花费在哪里？	• 把价格从 `Double` 类型改为 `BigDecimal` 有没有产生热点？ • 与应用的其他部分相比，哪种树操作最昂贵？

在第一次对指令簿 profile 的结果中，你现在了解到指令簿大量使用了 Scala 集合库，但是你对究竟是指令簿里的哪部分操作造成了性能问题还不太有感觉。为了加深对不同的指令簿操作的成本的理解，可通过查看 Call Tree 页来获取自顶向下的视图：

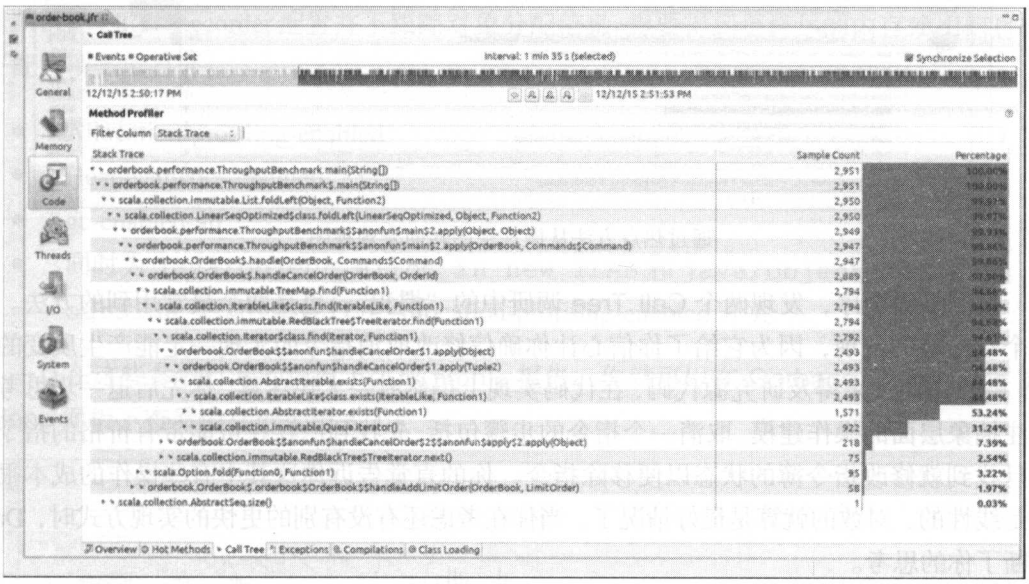


用调用树调查指令簿的瓶颈

沿着 Call Tree 下钻，一切都变得很清楚：取消操作是瓶颈。花在取消指令上的时间占

据压倒性的地位，远超增加指令花的时间。你打电话给公司最有眼力的交易员，Dave，告诉他你的发现。当你告诉 Dave 主要是取消操作开销太大时，他眼睛亮了起来。作为交易领域的专家，他非常清楚在市场波动期间，取消操作的频率比市场平静时会大幅增加。Dave 解释说交易员在市场波动期间频繁取消指令是为了快速调节应对价格变化。波动引发更多取消是因为交易员在过程中掌握到更有价值的价格信息。在你和 Dave 对话结束的时候，他让你取消所有别的在做的事情，专注于提高指令取消的性能。

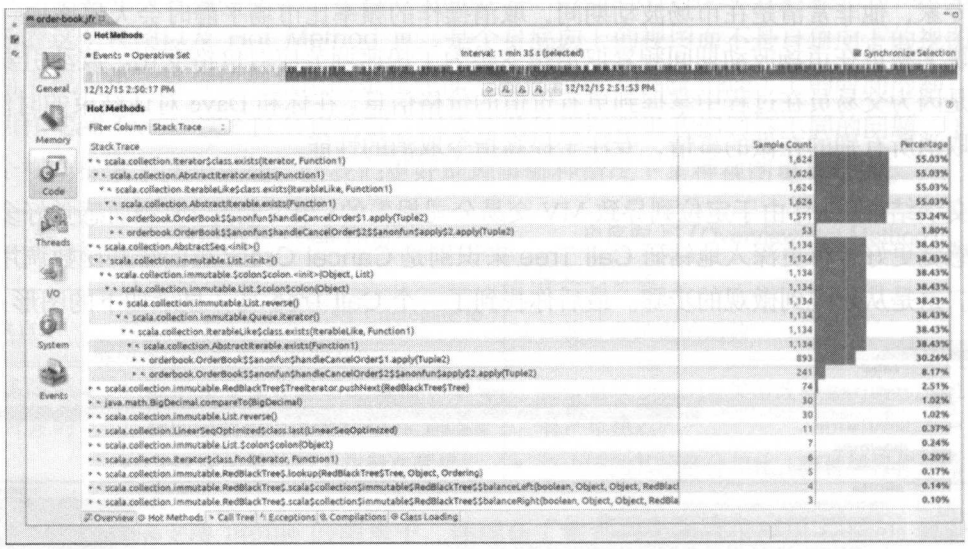
交谈结束后，你由于定位到导致 MV 交易公司损失的（很可能的）瓶颈而感觉好多了。为了感觉更好，你更深入地钻研 Call Tree 来识别是 Cancel Order 里的哪些函数调用特别昂贵。这是从宏观到微观的步骤。最后你找到了一个 Call Tree 页，显示如下的图形：



通过调用树理解哪个取消指令是最昂贵的

Call Tree 显示取消指令包含对 `RedBlackTree` 的 `find()` 函数和 `exists()` 函数的调用。当你更深入查看调用时，你同时注意到百分比列变小了。这是因为在自顶向下视图里，百分比列里显示的某个函数和它下级的函数的 CPU 时间总和。根据结果显示，84.48% 的 CPU 时间花在 `executingOrderBook$$anonfun$handleCancelOrder$1.apply()` 和它下面的函数上。从这视图上我们还能看到 84.48%的 CPU 时间、53.24%的时间花在 `withinAbstractIterator.exists()` 和它更深的函数调用上。这看上去像是最大的瓶颈，居第二位的则是 `Queue.iterator()`，占用了 31.24%的 CPU 时间。

有了这些信息，你很想开始自底向上研究，比如说，从最昂贵的函数开始，然后反推到对应的指令簿操作。为了满足好奇心，查看 Hot Methods 页，看到如下视图：



通过热点方法从微观视图转向宏观视图

按照量级顺序，发现两个 Call Tree 调研中的“惯犯”，也是两个最常用到的方法。这是个双保险的发现，因为它给了你信心让你确信修改取消指令的实现一定能产生质变的改进。由于你还没来得及研究源代码，在代码实现上仍然有一大块迷雾。你先后退一步思考，关注抽象层面的操作建模。取消一个指令的步骤包括，先找到已存在可能带有价格的指令，一旦找到就修改指令簿的状态以便移除指令。你的直觉告诉你这其中有些操作的成本很可能是线性的，对数的就算是最好情况了。当你在考虑还有没有别的更快的实现方式时，Dave 打断了你的思考。

你听到急促的声音，“你修好指令簿没有？我们现在就要把它部署上去！”当然你无从回答，而且想想第一天就要发布系统让你有点担忧。你和 Dave 分享了你的发现，希望能缓解他的进度压力，给你争取更多时间来思考。然而 Dave 听到指令簿问题仍未解决并不高兴，“我们每天都在为此亏钱！”你澄清自己已经明白形势的严重，正在以最快速度前进。毕竟这还是你的第一天！Dave 叹了口气，承认他有点过于严苛了，形势的恶化让他有点反应过度了。对话结束前，Dave 对你这么快上手表示欣赏，还跟你聊了会儿天，提到他难以理解他的新手机，有更多的内存，却还是很慢。“什么东西都快不起来！”他感叹道。当他提到内存的时候，你忽然灵光一闪。

你想起来你还没有查看内存使用报告，但愿有什么能通过调节垃圾搜集策略而不用改代码就能奏效的机会。在修改代码前，你开始检查 Memory 页组，看看能不能从内存分配模式中获得一些启发。

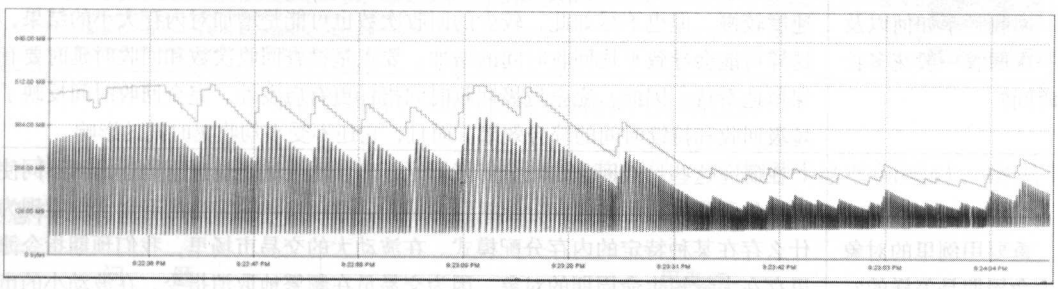
内存分配模式是什么？

Memory 页组是还剩的最后一个有待深入查看的分析结果了。尽管我们还没有花时间看指令簿的代码，Code 页组已经给我们展示了不同指令操作的相对成本了。研究 Hot Methods 给我们提供了用在指令簿的不同领域的对象的类型。查看内存分配模式时，我们希望识别出年轻代和老年代的垃圾回收趋势以及哪些对象分配得最少。

默认的 Flight Recorder 配置并不跟踪对象分配信息。想要得到更详尽的内存使用信息，以下配置项需要打开：

- Allocation-profiling-enabled
- Heap-statistics-enabled
- gc-enabled-all
- 同时为 `java/object_alloc_in_new_TLAB` 和 `java/object_alloc_outside_TLAB` 事件打开 Allocation-profiling-enabled 开关

一旦在这些参数打开的情况下生成 profile 报告，你就可以在 Garbage Collections 页的 Heap 面板下查看内存分配模式了：

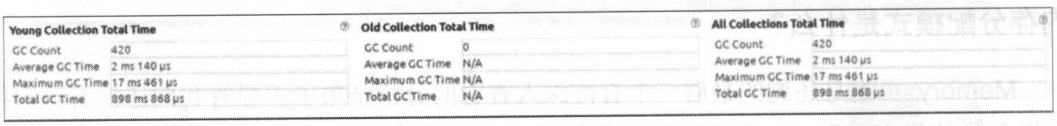


通过垃圾搜集面板使内存分配模式可视化

这个视图展示了一种通常称为锯齿模式的形状。当 JVM 在释放年轻代内存时，频繁的垃圾搜集产生了锯齿状的图形模式。垃圾搜集调节是一个大主题，超出了本书的范围。我们建议你通过阅读这篇极好的博客文章来深入理解这一主题，“*Understanding Java Garbage Collection*”（`http://www.cubrid.org/blog/dev-platform/understanding-`

java-garbage-col lection/)。

如下面的截图所示，Flight Recorder 在 GC Times 页下还提供了按照垃圾回收类型分类的总结指标：



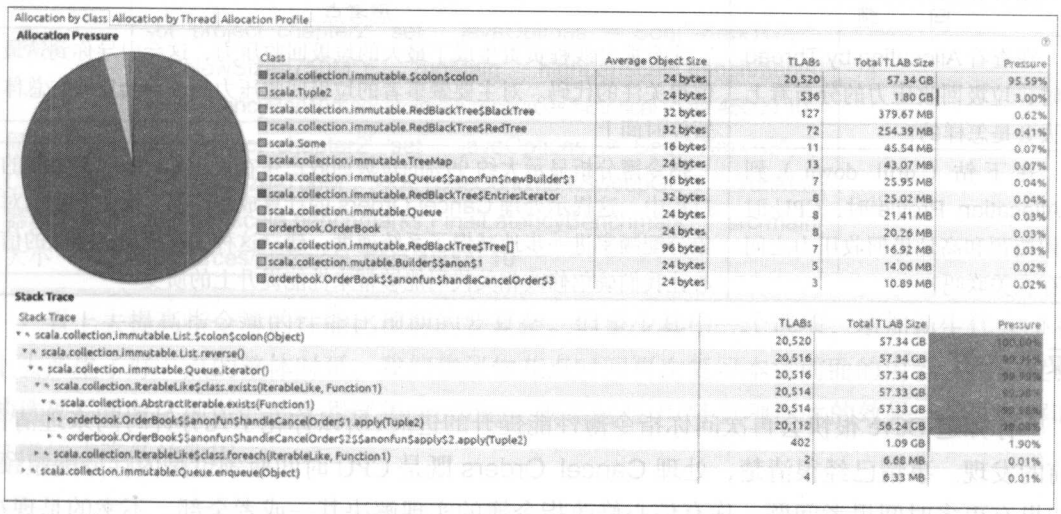
垃圾搜集事件类型摘要

下表列出了在查看可视化的堆栈使用图和按照搜集类型做的垃圾回收分解图时值得考虑的问题：

问　题	思考点
整体来说，内存使用是否保持平稳？还是斜向上或斜向下？	向上倾斜可能提示内存泄露。在这种场景下，堆内存逐渐增长，直到老生代填满，触发一次老生代回收。如果存在内存泄露，老生代回收无法完全释放内存，最终导致内存溢出错误。指令簿的内存使用情况下看上去在整个 profile 期间保持平稳。在做这种判断的时候，做尽可能长的 profile 以便尽可能看到全貌
异常暂停时间（outlier pause time）是否可其他主要事件有关联？	根据垃圾回收分解，最大回收时间比平均回收时间大一个数量级。查看 Heap 面板寻找远大于平均数的回收暂停时间。你是否在异常值中间找到某种模式？考虑你的应用和外部系统的交互以及其部署的机器，是否能给极端的暂停时间找到某种合理解释？另外，也值得在多次 profile 之间比较异常值，看看这种模式是只出现在某种场景下还是普遍存在
回收频率如何以及一次回收一般花多长时间？	在其他条件相同的情况下，回收次数越少越好，因为这可能表示垃圾产生的速度较慢。但也不尽如此，较少的回收次数也可能是增加对内存大小的结果，这样可能会导致平均回收时间的增加。要点是检查回收次数和回收时延时要有保留地看待。因此，完全垃圾回收时间指标更有启发性。完全回收时间反映了垃圾回收和持续时间的综合效果。而且，它还不受平均收集时间的影响
重要用例里的对象生命周期是怎样的？	在研究这些垃圾回收性能分解指标时，非常重要的是通过它构建出对不同使用场景下你的应用的内存分配机制的感觉。理解这种关联也许有助于你发现为什么存在某种特定的内存分配模式。在波动大的交易市场里，我们预期指令簿里存在大量短生命周期的对象，因为交易员在频繁地取消指令。在波动小的市场里，我们预期指令在指令簿里等待的平均时间会比较高，意味着更多的长生命周期对象

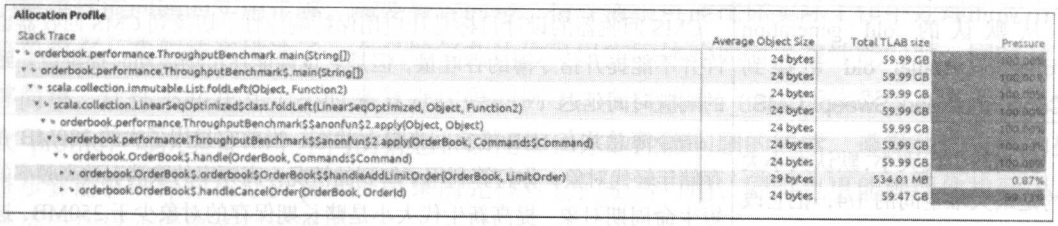
研究这些内存分配视图提供了内存分配活动方面的总结。Allocations 页提供了几个不同的方式来查看应用的哪些部分造成了较大的内存申请压力。Flight Recorder 提供了 3 种分

配视图：根据类、根据线程和根据 profile:



通过类内存分配视图找到与高压力的List分配的相关性

类和 profile 分配显示在上面的截图里。注意跳过了根据线程的分配，因为指令簿是单线程的。



通过自顶向下分配分析视图来确认内存分配压力

当检查这些内存分配视图时，应该考虑下面这些问题。当你阅读这些问题时，请思考要怎样回答这些问题来更好地理解如何提高指令簿的性能：

问　题	思考点
在查看 Allocation by class 时，回收压力大的类和关键路径上的类之间是否正相关？	如果你判断造成主要垃圾回收压力的类经常是在关键路径上分配，那你就有好的理由相信如果能够优化关键路径，就会同时获得算法速度和垃圾回收两方面的收益。指令簿的 profile 结果指出 List 分配是最大的问题点。回溯分析显示 List 的分配主要发生在 Cancle Orders 里，正是我们已经发现的瓶颈

续表

问　题	思考点
在查看 Allocation by Thread 时，垃圾回收压力的分布看上去会是怎样的？	注意那个线程负责生成了最大的垃圾回收压力，这会引导你到达最值得关注的代码。对主要肇事者的垃圾回收压力的改善会反映在总体暂停时间上
在下钻（drill down）到 Allocation Profile 时，CPU 时间瓶颈和高垃圾回收压力之间存在关联吗？	指令簿分析显示大约 99%的垃圾回收产生在处理 Cancel Orders 的过程中。这提示处理 Cancel Orders 的计算成本高昂，而由于高的对象分配频率进一步地减慢了系统速度。建立这种关联提供了强烈的信心让我们确信修改相关代码能够带来性能提升上的质变

反败为胜

你知道 Dave 很快会再次问你指令簿性能提升的进展，所以你花了几分钟时间来理清目前的发现。事情已经很清楚，处理 Cancel Orders 既是 CPU 时间瓶颈也是内存分配瓶颈。如果有更多时间思考问题，你有信心修改指令簿的实现解决其一或者全部。不幸的是现在缺少的正是时间。在内存分配方面观察到的一个有趣现象是在市场波动期间大部分垃圾回收对象都是短生命周期对象。两种低成本的选项浮现在你脑海里：

JVM 内存调节选项	假　设
从默认的 old generation collection,parallel old 切换到 Concurrent Mark Sweep（CMS）	CMS 回收器的设计目标是让应用保持响应性。切换到 CMS 回收器也许并不能提升指令簿的吞吐量，但是它也许能在市场波动时提供更一致的响应时间延迟
提高新生代大小，默认大小大约是最大堆空间的 1/4，把它改成最大堆空间的 3/4	指令簿总共有 1GB 堆空间来保存状态，当前它只用了大约 380MB 在存储年轻代对象。你打算利用了解到的情况：高频率的取消引致高频率的短生命周期对象。提高新生代大小是赌长期保存的对象少于 250MB，这样一来通过增加年轻代堆空间可以减少垃圾回收频率从而提高吞吐量

下表总结了每次实验的结果：

设　置	指　令	第 99 分位数（毫秒）
初始	sbt 'project chapter2' 'set javaOptions := Seq("-Xmx1G")' 'runMain highperfscala.benchmarks.FinalLatencyBenchmark src/main/resources/historical_data 45000 10'	92
使用 CMS 回收器	sbt 'project chapter2' 'set javaOptions := Seq("-Xmx1G", "-XX:+UseConcMarkSweepGC")' 'runMainhighperfscala.benchmarks.FinalLatencyBenchmark src/main/resources/historical_data 45000 10'	118

续表

设　置	指　令	第 99 分位数（毫秒）
750M 大小	sbt 'project chapter2' 'set javaOptions := Seq("-Xmx1G", "-XX:NewSize=750M")' 'runMain highperfscala.benchmarks.FinalLatencyBenchmark src/main/resources/historical_data 45000 10'	74
同时使用 CMS 回收器和 750M 大小	sbt 'project chapter2' 'set javaOptions := Seq("-Xmx1G", "-XX:NewSize=750M", "-XX:+UseConcMarkSweepGC")' 'runMain highperfscala.benchmarks.FinalLatencyBenchmark src/main/resources/historical_data 45000 10'	148

看上去提高指令簿的性能比预期的要复杂。但至少其中一个选项，增加新生代大小看上去产生了好一点的整体时延。我们建议你花点时间再次复习本章前面的内容，在设置新的选项后重复之前做的基准测试和 profiling。观察这些 JVM 选项产生的效果，尝试理解时延增加或降低的原因。

提醒一句

需要花点时间来强调我们前面解释的 profile 结果的上下文。我们走查了一个例子，这个例子反映了真实世界应用的多方面考虑因素。我们展示了本书作者在日常工作中应用过的解决性能问题的实用步骤。需要警告的是，指令簿很可能比你实际工作中处理的应用要简单很多。我们特意挑选了一个足够复杂让你可以实际体验如何走查性能问题，同时又足够简单，让你不必花几个小时来复查代码的例子。在实践中，很可能你需要重复运行很多次 profile，每次验证一个不同的猜想，以便找到你的性能问题的线索。遵循我们有序的排查步骤能够确保你在分析 profile 结果前先检查它们是否可用，同时也更能确保你找到用来精确定位修改位置的可靠线索。

profiling 检查清单

在本章里，我们走查了 profiling 检查清单上的每一项。在此提供一份完整的清单方便参考，如下：

1. 是否在预期的资源环境下测试？
2. 在 profiling 期间系统环境是否干净？
3. JVM 内部资源配置是否符合期望？

4. CPU 瓶颈在哪里？

5. 内存分配模式是什么？

微基准取得大进步

在接下来的章节里，我们将与你分享来自函数式编程范式和 Scala 编程语言的技巧，使你能写出高性能的软件。但是你不应该简单地照搬我们的药方。度量性能是判断我们的修改是否能提升性能的前提。微基准是一个术语，用来描述一个测量大应用里面的一小块孤立单元的性能的基准测试。由于微基准设计来测试一小块代码，当我们做了一些很小的修改的时候，跑微基准测试一般要比对整个应用做基准测试要容易一些。

不幸的是，准确地观察小修改造成的性能改变是不容易的事，特别是在 JVM 上。考虑指令簿相关的几个适合于微基准的修改例子。

- 把保存待处理的限制指令的数据结构替换成能更有效处理取消指令的数据结构。
- 把股票价格从 double 类型正规化成整数类型表示以便降低匹配的开销。
- 根据访问频率对一组分支语句的分支顺序做调整，判断对性能有多大的提升。

你如何在修改前后度量性能？你可能会尝试写一些小的基准测试程序，类似于 `ThroughtputBenchmark` 这种。这个步骤可能会给你提供不可信的测试结果，因为 JVM 具有一定的智能。JVM 采用一组启发式优化来做运行时优化。在生产环境下，这些优化是好的，因为性能提升总是好事。但是在微基准测试中，这些优化是有害的，因为它使你不确信你的微基准是否只隔离了你关心的小的代码修改。JVM 能够智能执行的优化包括以下这些：

- 死代码删除（Dead-code elimination）
- 即时编译（参考前文提到过的 JIT 编译器）
- 常量合并（如果一个函数的结果只依赖于测试，而对这个函数的调用有恰巧是个常量，则可以优化避免反复计算）

我们推荐你通过阅读 Oracle 的 *The Java Hotspot performance engine architecture* 了解更多关于 JVM 优化的知识（`http://www.oracle.com/technetwork/java/whitepaper-135217.html`）。

考虑到隔离小块代码如此困难，我们要怎么才能写出正确的微基准测试呢？幸运的是，

OpenJDK 团队注意到了这些挑战并因此推出了一个解决问题的库，叫作 JMH，Java microbenchmarking harness（http://openjdk.java.net/projects/code-tools/jmh/）。

JMH 正是设计来解决我们前面描述的困难，来帮助我们隔离代码修改的性能影响。使用 JMH 的过程类似于使用其他测试库。类似于 JUnit，JMH 定义了一组注解来控制测试配置和执行。尽管有多种方式来运行测试，我们专注于用 sbt 插件——sbt-jmh 插件 https://github.com/ktoso/sbt-jmh 来运行测试，以简化使用。我们来走一遍创建、运行和分析指令簿的微基准测试的过程。在后面的章节里，我们会发挥 JMH 知识，有目的地度量修改的效果。



你最近对手头的应用有什么修改能用上微基准的好处吗？如果你没试过对修改做微基准，是否认为微基准可能会把你导向不同的解决方案？

对指令簿做微基准

通过调节 JVM 内存设置取得一些性能提升后，你把眼光放到了更好地理解取消操作的性能上。基于 scala.collection.immutable.Queue 的存在，你猜测也许有一种遍历成本是线性的先进先出队列能更好地支撑指令取消。验证猜想的一种方法是设计一个微基准来度量不同场景下的取消操作性能。你头脑风暴了如下场景：

- 取消不存在的指令
- 取消某个价格水平的第一条指令
- 取消某个价格水平的最后一条指令

取消不存在的指令这种场景是真实存在的，当一个待成交的指令在取消指令到达前达成了交易时就会发生。这个场景很有意思，因为你不确定是否有某种提前终止逻辑来降低这个操作的开销，还是说取消一个不存在的指令需要遍历整个指令簿。另外两个场景专注于证券交易所提供的指令填充确保（fill guarantee）。当多个相同价格的指令被下达时，交易所确保它们按照先到达先服务的规定处理。猜测你在 profiling 结果里看到的 FIFO 队列保存了相同价格指令的时间顺序。在线性因素下你预期取消队列里的第一条指令要比取消最后一条指令快。

在读完优秀的 JMH 例子（http://hg.openjdk.java.net/code-tools/jmh/

file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/)并做了深入思考之后，你终于写出了一个测试来捕捉你所感兴趣的场景。以下是完整代码，后面有个走读过程，代码显示如下：

```
@BenchmarkMode(Array(Throughput))
@OutputTimeUnit(TimeUnit.SECONDS)
class CancelBenchmarks {

  @Benchmark
  def cancelLastOrderInLine(b: BookWithLargeQueue): (OrderBook, Event)
=OrderBook.handle(b.book, b.cancelLast)

  @Benchmark
  def cancelFirstOrderInLine(b: BookWithLargeQueue): (OrderBook, Event)
=OrderBook.handle(b.book, b.cancelFirst)

  @Benchmark
  def cancelNonexistentOrder(b: BookWithLargeQueue): (OrderBook, Event)
=OrderBook.handle(b.book, b.cancelNonexistent)
}

object CancelBenchmarks {

  @State(Scope.Benchmark)
  class BookWithLargeQueue {
    private val p = Price(BigDecimal(1.00))
    private val firstId: Int = 1
    private val defaultCancelLast = CancelOrder(OrderId(-1))

    @Param(Array("1", "100", "1000"))
    var enqueuedOrderCount: Int = 0

    var book: OrderBook = OrderBook.empty

    @Setup(Level.Trial)
    def setup(): Unit = {
      if (enqueuedOrderCount < 0)
```

```
      sys.error(s"Invalid enqueued order count = $enqueuedOrderCount")
    assert(book == OrderBook.empty)
    assert(cancelLast == defaultCancelLast)

    cancelLast = CancelOrder(OrderId(enqueuedOrderCount))
    book = {
      (firstId to enqueuedOrderCount).foldLeft(OrderBook.empty) {
        case (ob, i) =>
          OrderBook.handle(ob, AddLimitOrder(BuyLimitOrder(OrderId(i),
          p))).1
      }
    }

    assert(cancelLast != defaultCancelLast)
    if (enqueuedOrderCount > 0)
      assert(book.bids.head._2.size == enqueuedOrderCount,
        s"Book built incorrectly! Expected book to contain " +
          s"$enqueuedOrderCount bids for $p, but actual book is $book")
  }

  var cancelLast: CancelOrder = defaultCancelLast
  val cancelFirst: CancelOrder = CancelOrder(OrderId(firstId))
  val cancelNonexistent: CancelOrder = CancelOrder(OrderId(-1))
 }
}
```

这里就不重复 JMH 文档里的内容了，我们专注于特别有意思的部分，建议你自己去查看 JMH 样例了解更多上下文。查看 `CancelBenchmarks` 类，可以看到用注解定义了基准测试并控制测试结果的输出。有几种可用的基准测试模式。我们用吞吐量模式来度量在固定期间内基准完成的次数。每个取消基准测试的只在被取消的指令的 ID 上有区别。我们转换注意力到 `CancelBenchmarks` 对象，该类提供了必要的脚手架（scaffolding）来帮助我们设置每个基准测试。

`CancelBenchmarks` 对象定义了 `BookWithLargeQueue` 状态，我们观察到它是传给每个基准测试的参数。定义每个测试所需要的状态是达到把基准测试参数化这个目标的第一步。为了简化这组测试，我们创建了一个只有一种价格水平，也就是 1 美元的指令簿。

我们专注于清扫（sweeping）队列里积压的 1 美元价格的指令以帮助识别运行时行为，我们相信应该是按线性时间操作的。param 注解提供了一组默认值用来清洗队列里的指令。我们用 setup 注解来指示 JMH 在调用每个基准测试前先准备指令簿状态。对每个不同条数的指令队列，JMH 调用 setup 方法来创建一个包含理想数量的待处理指令的价格水平位于 1 美元的指令簿。

接着，我们用 sbt 来运行基准测试。JMH 提供了很多命令行参数来控制测试配置项，可以用如下的 sbt 指令来查看这些配置参数：

```
sbt 'project chapter2' 'jmh:run -help'
```

所有通过注解配置的参数都可以通过提供相应的命令行标志来覆盖掉。下面是调用 CancelBenchmarks 的示例：

```
sbt 'project chapter2' 'jmh:run CancelBenchmarks -wi 3 -w 5s -i 30 -r
10s -jvmArgs "-Xmx1G -Xms1G" -gc true -foe true -p
enqueuedOrderCount=1,10,50,100'
```

在这次调用中，我们配置 3 个预热循环，每个运行 5 秒钟。预热循环不计入吞吐量结果中。我们配置了 30 个要记录的循环，每个持续 10 秒钟，用来计算吞吐量。我们给测试提供了 1GB 堆内存大小，打开当遇到未捕获异常时退出基准测试的选项，以便对代码回归做保护。最后，我们参数化了我们希望清洗的队列指令数量，指示我们要对在数量分别为 1、10、50、100 个指令队列运行 3 个预热循环和 30 个记录循环。

对于地址簿里只有一条指令的场景，我们猜测所有指令的开销应该是一样大的。因为我们认为取消指令的开销是线性增长的，我们期望 10 条指令计数器的队列的每个基准测试应该比 50 条指令的大约慢 5 倍。我们把队列上限定在 100，因为在跟 Dave 的讨论中，我们了解到以他的经验来看，他从未碰到过一个地址簿里相同价格的指令超过 85 个的情况。把上线定在 100 确保我们理解的性能特性是不太会在生产环境下遇到但是理论上可能发生的。



假设你在为你的系统的性能最敏感的用例编写微基准测试。为了完整理解系统是怎么执行该用例的，有哪些变量是你觉得重要的？你要怎样识别一个基础用例、步进值以及最大值来参数化你的测试？考虑和领域专家讨论或者使用生产数据来指导你的决策。

在执行测试后，我们看见如下的结果总结：

基准测试	加入队列的指令条数	吞吐量/每秒操作数	错误/每秒操作数	错误（吞吐量的比率）
Cancel first order	1	6,688,878.23	Â±351,518.041	Â±5.26
Cancel first order	10	2,202,233.77	Â±103,557.824	Â±4.70
Cancel first order	50	555,592.56	Â±18,632.547	Â±3.35
Cancel first order	100	305,615.75	Â±14,345.296	Â±4.69
Cancel last order	1	7,365,825.52	Â±284,773.895	Â±3.87
Cancel last order	10	1,691,196.48	Â±54,903.319	Â±3.25
Cancel last order	50	509,339.60	Â±15,582.846	Â±3.06
Cancel last order	100	242,049.87	Â±8,967.785	Â±3.70
Cancel nonexistent order	1	13,285,699.96	Â±374,134.340	Â±2.82
Cancel nonexistent order	10	3,048,323.44	Â±140,983.947	Â±4.62
Cancel nonexistent order	50	772,034.39	Â±16,535.652	Â±2.14
Cancel nonexistent order	100	404,647.90	Â±3,889.509	Â±0.96

从这个结果集，我们可以回答一些有意思的问题来帮助我们掌握指令簿的性能特性。这里是期望你在分析结果后能够回答的问题：

- 只包含一条指令的指令簿基础测试用例是否能够定性三个基准测试具有一致的性能？
- 随着队列里的指令数量增长，基准测试是否表现出线性的吞吐量下降？
- 随着队列里的指令数量增加，不同基准测试间是否出现性能差异（比如说，当队列里的指令数量为 100 而不是 10 的时候，取消第一条指令和取消最后一条指令的性能是否有差别）？
- 能否看得出在取消不存在的指令时是否存在提前终止逻辑？

除了这些特定于指令簿案例的问题外，对于任何基准测试结果，有必要拷问自己下面的问题：

- 这个测试结果是否合理？
- 可能搞错了什么东西（产生了错误的结果），我们能否采取什么措施防止这些错误发生？

度量或测试设置上的错误会破坏测试结果的完整性。这些问题的目的是提醒你用挑剔

的眼光审视微基准结果。合理性检测要求你为执行的测试开发一种思维模型。对于指令簿来说，我们需要考虑每秒发生的取消操作的合理数量是什么范围。回答这个问题的方法之一是拿到一个吞吐量测试结果，比如说取消 50 个指令的队列里的最后一条指令，然后计算平均每个操作花的时间。这样做是有效的，因为在之前的基准测试里，我们对每操作级别的消耗已经有了认识；509339.595 对应大约 0.002 毫秒每操作。这个结果看上去低得令人吃惊，但是要考虑到这些结果不包含协同遗漏，因为没有目标吞吐率（也就是说，测试试图每秒钟发送尽可能多的取消指令）。成本这么低的另一个可能原因是指令簿里只有一个价格水平。典型的指令簿里包含买卖双方的各种价格。这可能引导我们设计好不同价格水平的基准测试，以更好地理解管理不同价格水平的开销。

第二个问题迫使我们挑剔地分析测试设置和测试方法论。比如说，我们怎么知道 setup 函数是否生成了期望的指令簿？防范这种错误的一种方法是加入断言语句来确保符合约束。另一个要顾虑的问题是验证基准测试里的每一条取消操作是否产生了期望的事件。为单独一条指令运行加个断言或许能消除这个顾虑，但是把断言留在大批量的基准测视里很可能影响性能，所以必须慎用。为了增加安全性，可以适时地为被测场景写一些单元测试来确保功能正确，确保单元测试和性能测试代码是共享的。

在解释 JMH 结果时，考虑计算错误的显著性是很重要的。JMH 根据基准测试循环构建一个置信区间（confidence interval）用来计算错误。置信区间假设结果遵循常规分布，错误代表计算出来的 99.9%执行区间的范围（the error represents the range of the computed 99.9% confidence interval）。这代表着当所有其他条件相同时，运行更多的基准测试循环可以提高结果的可信度。结果表的最后一列显示结果的可变性。可变性越低，结果就越值得采信。高的结果可变性提示在度量方法上存在某种错误或者有某种东西影响了度量正确的性能特征的能力。这通常是一种警告，提示你重新研究你的测试方法论，并且你不应该太信任这个测试结果。



拿我们的例子来说，我们运行了 30 个循环来记录吞吐量信息。如果少运行几个循环你觉得会怎样？另外，考虑运行较少的循环次数同时增加持续时间。比如 10 个循环，每个持续 30 秒。做个猜测，然后运行 JMH 来看看结果如何。对于不同的基准测试参数建立敏感性是另一种方法来对将来的基准测试步骤建立直觉的方法。

由于我们的 JMH 配置没有考虑协同遗漏，而是给指令簿发送了一个取消指令的消防带（firehose），我们应当专注于相对结果而不是绝对的吞吐量数值。基于相对差异的关于指令簿的测试结果应该独立于测试环境（比如可用核数或内存大小）。专注于相对值有其价值所在，因为该结果对于修改更稳定。如果将来的代码修改造成显著的相对变化，比如指数级而不是线性的取消指令性能下降，你可以高度自信这个性能下降是代码修改造成的而不是环境变化引起的。

在本节里，我们看到了如何设置、执行和解释 JMH 微基准测试。在过程中，我们了解了不使用 JMH 做微基准存在哪些缺陷，以及在基准测试结果分析时需要关注哪些因素。我们只是粗略看了一下 JMH 最表层的能力。 我们还会在后面的章节里进一步介绍 JMH。

小结

恭喜，在本章里，你帮助提升了 MV 交易公司的性能，而直接增加了公司的利润，减少了损失！在过程中，你深入了解了如何在 JVM 上做基准测试和 profile，以及有哪些缺陷要避免。你还学习了 JMH 微基准初阶，它使你可以有目的地评估后续章节的性能提升。在下一章里，我们将看看如何用 Scala 语言特性来写函数式软件，会用本章学到的技术来评估其性能影响。

3

释放 Scala 的性能

在本章，我们来学习 scala 特定的组件和语言特性，检查它们对性能有哪些帮助或妨碍。掌握了新学到的性能度量知识后，我们来分析如何更好地利用 Scala 编程语言提供的丰富特性。对每种特性，我们先介绍它，然后展示它编译成字节码是什么样的，然后识别使用该特性时有哪些危险和其他考虑事项。

在本章里，我们将展示 Scala 源代码和编译器生成的字节码。有必要这样做来加强你对 Scala 和 JVM 交互机制的理解，这样你才能对软件的运行时表现建立一种洞察。我们将使用 javap Java 反汇编器来反编译生成的字节码，命令如下：

```
javap -c <PATH_TO_CLASS_FILE>
```

-c 开关打印反汇编出来的代码。另一个有用的选项是-private，用来打印私有方法的字节码。要了解更多 javap 的用法，请参考使用手册。我们将讨论的例子不需要很深入的 JVM 字节码知识，不过如果你又想去学习更多关于字节码的知识，请参考 Oracle 的 JVM 规范：http://docs.oracle.com/javase/specs/jvms/se7/htm l/jvms-3.html#jvms-3.4。

偶尔，我们还会检查去除了 Scala 专门特性后的 Scala 源代码，用这个命令：

```
scalac -print <PATH>
```

这是一种有效的方式来看 Scala 编译器如何去糖化（desugar）Scala 的方便语法来构造 JVM 能够执行的代码。在本章里，我们将探讨如下主题：

- **值类（Value class）和标记类型（tagged types）**

- 专门化（Specialization）
- 元组
- 模式匹配
- 尾递归
- Option 数据类型
- Option 的替代品

值类

在第 2 章里，我们介绍了指令簿应用的领域模型，其中包含两个类，Price 和 OrderId。我们指出我们为 Price 和 OrderId 构造领域类是为了给被包装的 BigDecimal 和 Long 提供上下文信息。尽管这样做提高了代码可读性和编译期安全，但也增加了我们的应用创建的实例数量。分配内存和生成类实例带来了更多的垃圾回收压力，因为这样增加了回收的频率而且有可能引入额外的长生命周期对象。垃圾收集器不得不更努力地回收它们，这可能会严重影响应用的时延。

幸运的是，从 Scala2.10 开始，开发人员可以用 AnyVal 抽象类来自定义值类从而解决这个问题。AnyVal 类在 Scala doc（http ://www.scala-lang.org/api/current/#scala.AnyVal）里定义如下，“所有值类型的基类，用来定义不以对象形式在宿主系统（host system）里实现的值。”AnyVal 类可以用来定义值类，这种类型会受到编译器的特别对待。值类在编译器优化以避免分配实例而使用包装的基础类型。

字节码表示

举例来说，为了提高指令簿的性能，我们可以把 Price 和 OrderId 定义为值类：

```
case class Price(value: BigDecimal) extends AnyVal
case class OrderId(value: Long) extends AnyVal
```

为了展示值类所受到的特别对待，我们来定义一个假方法，接受 Price 值类和 OrderId 值类作为参数：

```
def printInfo(p: Price, oId: OrderId): Unit =
  println(s"Price: ${p.value}, ID: ${oId.value}")
```

根据这个定义，编译器生成了如下的方法签名：

```
public void printInfo(scala.math.BigDecimal, long);
```

可以看到编译器生成的签名接受的其实是 BigDecimal 和 long，而我们的 Scala 代码使用的是模型里定义的类型。这意味着我们不能用 BigDecimal 和 long 实例来调用 printInfo，因为编译器会报错。



有意思的是 printInfo 的第二个参数并未被编译成 Long（对象）而是 long（基础类型，注意首字母小写的 l）。Long 和其他对应于基础类型的对象，比如 Int、Float 或 Short 被编译器特殊处理在运行时用它们对应的基础类型来表示。

值类也可以定义方法。我们来给 Price 类型加点料：

```
case class Price(value: BigDecimal) extends AnyVal {
  def lowerThan(p: Price): Boolean = this.value < p.value
}

// Example usage
val p1 = Price(BigDecimal(1.23))
val p2 = Price(BigDecimal(2.03))
p1.lowerThan(p2) // returns true
```

我们新的方法用来比较两个 Price 实例的大小。在编译期，编译器为 Price 生成了一个伴生对象。这个伴生对象定义了 lowerThan 方法，接受两个 BigDecimal 对象作为参数。实际上，当我们在某个 Price 实例上调用 lowerThan 时，编译器把代码从实例方法调用转换成了伴生对象里定义的静态方法调用：

```
public final boolean lowerThan$extension(scala.math.BigDecimal,
scala.math.BigDecimal);
   Code:
      0: aload_1
      1: aload_2
      2: invokevirtual #56  // Method
scala/math/BigDecimal.$less:(Lscala/math/BigDecimal;)Z
      5: ireturn
```

如果用伪代码重写一下前面的 Scala 代码的话，其实是这样的：

```
val p1 = BigDecimal(1.23)
val p2 = BigDecimal(2.03)
Price.lowerThan(p1, p2) // returns true
```

性能考虑

值类是你的开发者工具箱的一种极好的补充。它们有助于减少实例数量，给垃圾回收器节约一点工作量，同时又允许我们使用能反映业务抽象的有意义的类型。但是，扩展 AnyVal 是需要符合一些条件的。比如说，值类只能有一个接受一个 public val 参数的基础构造器。而且，这个参数本身不能是值类。我们前面看到可以在值类里用 def 来定义方法，但是注意在值类里 val 和 var 都是不允许的。嵌套的类或对象定义也是不允许的。还有一个限制是值类型只能扩展符合特定要求的 trait——这个 trait 必须扩展 Any，只包含 def 成员，没有任何初始化动作。如果上述任何条件不符合，编译器会报错。除了前面列出的约束外，在一些特殊场景 JVM 会实例化值类型（而不是我们期望的不实例化）。这些场景包括执行模式匹配或运行时类型检测或把值类型赋值给数组。最后一个场景的例子如下：

```
def newPriceArray(count: Int): Array[Price] = {
  val a = new Array[Price](count)
  for(i <- 0 until count){
    a(i) = Price(BigDecimal(Random.nextInt()))
  }
  a
}
```

生成的字节码如下：

```
public
highperfscala.anyval.ValueClasses$$anonfun$newPriceArray$1(highperfscala
.an
yval.ValueClasses$Price[]);
   Code:
      0: aload_0
      1: aload_1
      2: putfield      #29  // Field
a$1:[Lhighperfscala/anyval/ValueClasses$Price;
```

```
        5: aload_0
        6: invokespecial #80  // Method
    scala/runtime/AbstractFunction1$mcVI$sp."<init>":()V
        9: return
    public void apply$mcVI$sp(int);
      Code:
        0: aload_0
        1: getfield      #29  // Field
    a$1:[Lhighperfscala/anyval/ValueClasses$Price;
        4: iload_1
        5: new           #31  // class
    highperfscala/anyval/ValueClasses$Price
        // omitted for brevity
       21: invokevirtual #55  // Method
    scala/math/BigDecimal$.apply:(I)Lscala/math/BigDecimal;
       24: invokespecial #59  // Method
    highperfscala/anyval/ValueClasses$Price."<init>":(Lscala/math/BigDecimal
;)V
       27: aastore
       28: return
```

注意 newPriceArray 调用了 mcVI$sp，这导致在第 5 条指令创建了 ValueClasses$Price 实例。

因为把一个只包含一个属性的 case class 转换成值类只要扩展 AnyVal 就搞定，所以我们推荐你尽量使用 AnyVal。开销很小，又能生成有利于垃圾回收性能的代码。要学习更多关于值类的知识、限制和使用场景，你可以在这里看到详尽描述：http://docs.scala-lang.org/overviews/core/value-classes.html。

标记类型——值类的一种替代品

值类很容易使用而且能产生很好的性能提升。但是由于它的约束条件很多，有些场景下无法使用。本节我们快速介绍一种有意思的替代品，叫作标记类型，这个特性是在 Scalaz 库里实现的（https://github.com/scalaz/scalaz）。



Scalaz 实现的标记类型是受另一个 Scala 库的启发，这个库叫作 shapeless。shapeless 库提供了写类型安全、通用代码（boilerplate）最少的工具。尽管我们不会讨论 shapeless，但是建议你自己去做进一步了解：https://github.com/milessabin/shapeless。

标记类型是另一种用来确保编译期类型检查而又不造成实例化实例开销的方法。它们依赖于 Scalaz 库里定义的 Tagged 结构化类型和@@类型别名：

```
type Tagged[U] = { type Tag = U }
type @@[T, U] = T with Tagged[U]
```

我们重写 Price 的部分代码来发挥标记类型的威力：

```
object TaggedTypes {
  sealed trait PriceTag
  type Price = BigDecimal @@ PriceTag
  object Price {
    def newPrice(p: BigDecimal): Price =
      Tag[BigDecimal, PriceTag](p)
    def lowerThan(a: Price, b: Price): Boolean =
      Tag.unwrap(a) < Tag.unwrap(b)
  }
}
```

我们快速地走读一下这段代码。我们定义了 sealed trait PriceTag，用来标记其他实例。接着创建了 Price 类型别名，定义为打了 PriceTag 标记的 BigDecimal 对象。Price 对象里定义了一些有用的方法，包括 newPrice 工厂方法用来给传入的 BigDecimal 打标记然后返回 Price 对象（也就是打了 PriceTag 标记的 BigDecimal 对象）。我们还实现了等价于 lowerThan 的方法。这个函数接受两个 Price 对象（也就是两个打了标记的 BigDecimal 对象），从标记里抽取出内容，也就是两个 BigDecimal，然后比较。

我们用新的 Price 类型重写与前面相同的 newPriceArray 方法（代码略，你可以参考附带的源代码），打印出来的字节码如下：

```
public void apply$mcVI$sp(int);
```

```
      Code:
         0: aload_0
         1: getfield      #29  // Field a$1:[Ljava/lang/Object;
         4: iload_1
         5: getstatic     #35  // Field
   highperfscala/anyval/TaggedTypes$Price$.MODULE$:Lhighperfscala/anyval/Ta
gge dTypes$Price$;
         8: getstatic     #40  // Field
   scala/package$.MODULE$:Lscala/package$;
        11: invokevirtual #44  // Method
   scala/package$.BigDecimal:()Lscala/math/BigDecimal$;
        14: getstatic     #49  // Field
   scala/util/Random$.MODULE$:Lscala/util/Random$;
        17: invokevirtual #53  // Method scala/util/Random$.nextInt:()I
        20: invokevirtual #58  // Method
   scala/math/BigDecimal$.apply:(I)Lscala/math/BigDecimal;
        23: invokevirtual #62  // Method
   highperfscala/anyval/TaggedTypes$Price$.newPrice:(Lscala/math/BigDecimal
;)Ljava/lang/Object;
        26: aastore
        27: return
```

这个版本里，尽管我们把 Price 赋值给数组，但是看不到实例化 Price 了。标记的 Price 实现有一些运行时类型转换的开销，但是我们预期转换的开销要小于实例分配和垃圾回收的开销。我们会在本章后面部分再来讨论标记的类型，用它来替代 Scala 标准库里广泛使用的一个工具：Option 类型。

专门化

为了理解专门化的意义，掌握对象装箱的概念非常重要。JVM 定义了一些基础类型（boolean、byte、char、float、int、long、short 和 double），这些基础类型分配在栈空间而不是堆空间。把数据放到泛型类型（generic type）时，比如 scala.collection.immutable.List，JVM 引用它的对象等价物而不是基础类型。在这个例子里，实例化的整数列表会是堆分配的对象而不是整数基础类型。把基础类型转换成对应的对象等价物的过程叫作装箱（boxing），反向过程叫作拆箱（unboxing）。装箱是

性能敏感编程时的重要考虑因素，因为装箱操作造成堆空间分配。在执行数值计算的性能敏感的代码里，装箱和拆箱的开销可以造成显著的性能下降。下面的例子展示了装箱的开销：

```
List.fill(10000)(2).map(_* 2)
```

用 `fill` 操作创建列表产生了 10 000 次整数对象的堆分配操作。在 `map` 里执行乘法产生了 10 000 次拆箱再加 10 000 次装箱来把乘运算的结果放到新列表里。从这个简单例子可以看出由于拆箱和装箱操作会给关键的数学运算部分造成多大的性能下降。

根据 Oracle 教程所述（`https://docs.oracle.com/javase/tutoria l/java/data/autoboxing.html`），Java 和 Scala 里的装箱是透明的。 也就是说，如果没有仔细的 profiling 或字节码分析，你很难发现在哪里支付了对象装箱的开销。为了改善这个问题，Scala 提供了一个特性，叫作专门化。专门化是指编译期为泛型类型生成专门用于某个基础类型而不是对象包装的版本。在运行时，编译器生成的版本（一般被称为某类的专门版本）被实例化。这个过程消除了基础类型装箱的开销，意味着你可以在保持手写的专门实现的性能的同时定义泛型抽象。

字节码表示

让我们来看个实际例子来更好地理解专门化的机制。考虑一个简单的、泛型的股票购买数量类型，如下：

```
case class ShareCount[T](value: T)
```

基于这个例子，假设预期的使用场景是交换 `ShareCount` 的整型和 `long` 型表示。以这个定义来说，实例化一个基于 `long` 的 `ShareCount` 实例产生装箱开销，如下：

```
def newShareCount(l: Long): ShareCount[Long] = ShareCount(l)
```

这个定义翻译成对应的字节码：

```
  Public
highperfscala.specialization.Specialization$ShareCount<java.lang.Object>
newShareCount(long);
    Code:
       0: new           #21  // class orderbook/Specialization$ShareCount
       3: dup
       4: lload_1
```

```
      5: invokestatic  #27  // Method
   scala/runtime/BoxesRunTime.boxToLong:(J)Ljava/lang/Long;
      8: invokespecial #30  // Method
   orderbook/Specialization$ShareCount."<init>":(Ljava/lang/Object;)V
     11: areturn
```

在前面的字节码里，很明显，在指令 5，基础类型 long 值在实例化 ShareCount 前被装箱。通过引入@specialized 注解，我们可以通过让编译器生成 ShareCount 专用于基础类型 long 的版本，从而消除装箱。可以通过提供一组类型来指定你希望专门化的类型。根据 Specializables 特质（http://www.scala-lang.org/api/current/index.html#scala.Specializable）的定义，你可以为所有的 JVM 基础类型做专门化，此外还包括 Unit 和 AnyRef。以我们的例子来说，为整数和 long 专门化 ShareCount。

```
case class ShareCount[@specialized(Long, Int) T](value: T)
```

这样定义后，字节码变成这样：

```
  public
highperfscala.specialization.Specialization$ShareCount<java.lang.Object>
newShareCount(long);
    Code:
      0: new           #21  // class
highperfscala.specialization/Specialization$ShareCount$mcJ$sp
      3: dup
      4: lload_1
      5: invokespecial #24  // Method
highperfscala.specialization/Specialization$ShareCount$mcJ$sp."\<init>":
(J)V
      8: areturn
```

装箱操作不见了，替之以一个奇怪的类名，ShareCountmcJsp。这是因为我们调用了编译期生成的 ShareCount 版本，专门用于 long 值。通过检查 javap 的输出，我们可以看到编译器生成的专门化类是 ShareCount 的子类：

```
public class highperfscala.specialization.Specialization$ShareCount$mcI$sp
extends highperfscala.specialization.Specialization$ShareCount<java
.lang.Object>
```

当我们转向性能考虑一节时，请记住专门化的实现细节。在更复杂的用例中，专门化对继承的使用是需要取舍的一个考虑点。

性能考虑

初看上去，专门化好像是 JVM 装箱问题的解药。然而，在使用专门化时有几个要点需要考虑。随意使用专门化可能会显著增加编译时间和字节码大小。考虑对 Function3——接受三个参数返回一个结果——做专门化。对所有 4 个参数（译者注：3 个参数加一个返回值）为全部类型做专门化（Byte、Short、Int、Long、Char、Float、Double、Boolean、Unit 和 AnyRef）产生 10 的 4 次方也就是 10 000 种可能的排列组合。为此，标准库很少使用专门化。对你自己的使用场景，需要仔细考虑你希望为哪些类型做专门化。如果对 Function3 我们只为 Int 和 Long 做专门化，生成的类的数量减少到 2 的 4 次方，也就是 16 个。专门化对继承的使用需要额外的关注，因为很容易在扩展一个泛型类的时候丢失专门化。考虑如下例子：

```
class ParentFoo[@specialized T](t: T)
class ChildFoo[T](t: T) extends ParentFoo[T](t)

def newChildFoo(i: Int): ChildFoo[Int] = new ChildFoo[Int](i)
```

在这个场景里，你可能以为 ChildFoo 定义为支持基础整型，然而由于 ChildFoo 没有用@specialized 注解标注，并没有生成专门化类型。下面的字节码证实了这一点：

```
  public
highperfscala.specialization.Inheritance$ChildFoo<java.lang.Object>
newChildFoo(int);
    Code:
      0: new           #16  // class
highperfscala/specialization/Inheritance$ChildFoo
      3: dup
      4: iload_1
      5: invokestatic  #22  // Method
scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      8: invokespecial #25  // Method
highperfscala/specialization/Inheritance$ChildFoo."<init>":(Ljava/lang/O
bject;)V
```

```
    11: areturn
```

符合逻辑的改进是给 ChildFoo 加上@specialized 注解。在修改的时候，我们困惑地看到编译器对此给出了警告：

```
class ParentFoo must be a trait. Specialized version of class ChildFoo will
inherit generic highperfscala.specialization.Inheritance.ParentFoo[Boolean]

class ChildFoo[@specialized T](t: T) extends ParentFoo[T](t)
```

编译器指出你弄出了菱形继承问题，专门版本的 ChildFoo 同时继承了 ChildFoo 和 ParentFoo 的相应的专门版本。可以通过用 trait 特质建模来解决这个问题：

```
trait ParentBar[@specialized T] {
  def t(): T
}

class ChildBar[@specialized T](val t: T) extends ParentBar[T]

def newChildBar(i: Int): ChildBar[Int] = new ChildBar(i)
```

这个定义编译生成了专门版本的 ChildBar，符合我们的初始预期：

```
public
highperfscala.specialization.Inheritance$ChildBar<java.lang.Object>
newChildBar(int);
   Code:
      0: new           #32  // class
highperfscala/specialization/Inheritance$ChildBar$mcI$sp
      3: dup
      4: iload_1
      5: invokespecial #35  // Method
highperfscala/specialization/Inheritance$ChildBar$mcI$sp."<init>":(I)V
      8: areturn
```

一个类似的而且同样容易出错的场景是针对专门类型定义的泛型方法。考虑如下定义：

```
class Foo[T](t: T)
```

```
object Foo {
  def create[T](t: T): Foo[T] = new Foo(t)

}

def boxed: Foo[Int] = Foo.create(1)
```

这里，create 的定义类似于继承例子里的子类。包装了在 create 方法里实例化的基础类型的 Foo 包装实例将会被装箱。下面的字节码展示了 boxed 导致的堆空间分配：

```
public
highperfscala.specialization.MethodReturnTypes$Foo<java.lang.Object>
boxed();
  Code:
    0: getstatic     #19  // Field
highperfscala/specialization/MethodReturnTypes$Foo$.MODULE$:Lhighperfsca
la/
specialization/MethodReturnTypes$Foo$;
      3: iconst_1
      4: invokestatic  #25  // Method
scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      7: invokevirtual #29  // Method
highperfscala/specialization/MethodReturnTypes$Foo$.create:(Ljava/lang/O
bje
ct;)Lhighperfscala/specialization/MethodReturnTypes$Foo;
     10: areturn
```

解决方法是在调用点应用@specialized 注解，如下：

```
def createSpecialized\[@specialized T](t: T): Foo\[T] = new Foo(t)
```

最后一个有意思的场景是当专门化用于多个类型，而其中一个类型扩展了 AnyRef 或是个值类的时候。为了展示这个场景，考虑如下例子：

```
case class ShareCount(value: Int) extends AnyVal
case class ExecutionCount(value: Int)

class Container2[@specialized X, @specialized Y](x: X, y: Y)
```

```
def shareCount = new Container2(ShareCount(1), 1)

def executionCount = new Container2(ExecutionCount(1), 1)

def ints = new Container2(1, 1)
```

在这个例子里，你预期哪几个方法会装箱 Container2 的第二个参数？为了简要起见，我们省略了字节码，但是你自己可以很容易地去核实一下。结果发现 shareCount 和 executionCount 对整型做了装箱。编译器没有生成能够接受基础整型和扩展 AnyVal 的值（比如说 ExecutionCount）的专门版本。shareCount 也发生了装箱因为编译器先移除了值类型的类型信息。对于这两种情况，绕过的方法是对一组类型专门定义 case class（比如说 ShareCount 和 Int）。移除泛型使编译器可以选择基础类型。

从这些例子可以得出结论，对于没有装箱操作的应用，专门类型需要额外关注。因为编译器无法推论哪些场景下你是忘了写@specialized 注解，因此也就无法给出警告。这就需要你负责谨慎小心的 profiling 和检查字节码来检测是否专门化被意外丢弃了。



为了应对专门化的一些缺点，有一个编译器插件正在积极开发中，叫作 miniboxing，在 http://scala-miniboxing.org/。这个编译器插件采取了另一种不同的策略，它把所有基础类型编码成 long 型值并且携带元数据用来回溯原始类型。比如说 boolean 可以表示为 1 个字节的 long 属性来代表 true 或 false。用这种方法，性能类似于专门化而生成的类在排列组合量大的情况下要少几个数量级。另外，miniboxing 可以更稳健地处理继承场景，而且能在装箱发生时给出警告。尽管专门化和 minibox 的实现迥异，它们的使用方式十分类似。和专门化一样，你必须添加合适的注解来激活 miniboxing 插件。想学习更多关于这个插件的知识，你可以看 miniboxing 项目网站上的教程。

因为能在性能敏感代码上获得性能提升，值得花精力确保专门化机制生成无须堆分配的代码。为了让你理解专门化的价值，考虑下面这个对通过把股票数量和交易价格相乘来计算交易成本的运算做的微基准。为了简化，我们直接用了基础类型而不是值类。当然，在生产代码里是永远不会这么做的。

```
@BenchmarkMode(Array(Throughput))
@OutputTimeUnit(TimeUnit.SECONDS)
@Warmup(iterations = 3, time = 5, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 30, time = 10, timeUnit = TimeUnit.SECONDS)
@Fork(value = 1, warmups = 1, jvmArgs = Array("-Xms1G", "-Xmx1G"))
class SpecializationBenchmark {

  @Benchmark
  def specialized(): Double =
    specializedExecution.shareCount.toDouble * specializedExecution.price

  @Benchmark
  def boxed(): Double =
    boxedExecution.shareCount.toDouble * boxedExecution.price

}

object SpecializationBenchmark {
  class SpecializedExecution[@specialized(Int) T1, @specialized(Double)
T2](
    val shareCount: Long, val price: Double)
  class BoxingExecution[T1, T2](val shareCount: T1, val price: T2)
  val specializedExecution: SpecializedExecution[Int, Double] =
    new SpecializedExecution(101, 2d)
  val boxedExecution: BoxingExecution[Long, Double] = new
BoxingExecution(101, 2d)
}
```

在这个基准测试里，定义了两个版本的泛型成交类。

在计算总成本时 SpecializedExecution 完全没有装箱成本，而 Boxing Execution 需要对象装箱和拆箱来执行计算。用如下参数调用微基准：

```
sbt 'project chapter3' 'jmh:run SpecializationBenchmark -foe true'
```



我们用代码里的类级别的注解来配置 JMH。这跟我们在第 2 章里用命令行参数不同。注解的优势是可以为基准测试设置合适的默认

值，简化命令行调用。仍然能够用命令行参数来覆盖掉注解配置。我们用-foe 命令行参数打开遇错跳出（fail on error）开关，因为没有注解能设置这个开关。在本书后面的部分，我们都用注解来参数化 JMH。因为我们总是用相同的配置，所以在代码例子里就省略了。

结果总结在下表中：

基准测试	吞吐量（ops）	错误率（占吞吐量百分比）
boxed	251,534,293.11	±2.23
specialized	302,371,879.84	±0.87

微基准测试指出专门化版本提高了大约 17%的吞吐量。通过在关键代码里消除装箱，专门化带来了数量级的性能提升。对于性能敏感的运算，这个基准测试提供了理由让我们投入更多精力来确保专门化确实被正确地应用了。

元组

在 Scala 里元组是一等公民，这简化了需要把几个值打包在一起的场景。通过元组，你可以优雅地用简洁的语法返回多个值而无须定义 case class。 下一节展现了编译器是如何编译 Scala 元组的。

字节码表示

让我们来看看 JVM 是如何处理元组的创建的，以便理解 JVM 能如何更好地支持元组。为了建立直观感受，考虑创建任意两个元素的元组，如下：

```
def tuple2: (Int, Double) = (1, 2.0)
```

对应的字节码如下：

```
public scala.Tuple2<java.lang.Object, java.lang.Object> tuple2();
  Code:
    0: new           #36  // class scala/Tuple2$mcID$sp
    3: dup
    4: iconst_1
```

```
         5: ldc2_w        #37  // double 2.0d
         8: invokespecial #41  // Method scala/Tuple2$mcID$sp."\<init>":(ID)V
        11: areturn
```

字节码显示编译器把括号形式的元组定义语法去糖变成分配一个叫作 Tuple2 的类。每个不同参数个数的元组都有一个对应的 tuple 类(比如 Tuple5 支持 5 个元素的元组), 最多到 Tuple22。字节码的第 4 和第 5 指令还显示分配 tuple 实例使用的是基础类型版本的 Int 和 Double。

性能考虑

在上面的例子里，Tuple2 通过对两个泛型类型做专门化避免了基础类型的装箱成本。由于 scala 的元组语法很方便，把多个值组合在一起通常很方便。然而这样会导致额外的内存分配，因为大于两个元素的元组是没有专门化的。下面例子展示了这一点：

```
def tuple3: (Int, Double, Int) = (1, 2.0, 3)
```

这个例子类似于前面看过的元组定义，唯一不同的是现在有 3 个元素。这个定义生成了如下字节码：

```
    public scala.Tuple3<java.lang.Object, java.lang.Object,
java.lang.Object>
  tuple3();
    Code:
       0: new           #45  // class scala/Tuple3
       3: dup
       4: iconst_1
       5: invokestatic  #24  // Method
  scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
       8: ldc2_w        #37  // double 2.0d
      11: invokestatic  #49  // Method
  scala/runtime/BoxesRunTime.boxToDouble:(D)Ljava/lang/Double;
      14: iconst_3
      15: invokestatic  #24  // Method
  scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      18: invokespecial #52  // Method
```

```
  scala/Tuple3."<init>":(Ljava/lang/Object;Ljava/lang/Object;Ljava/lang/Ob
ject;)V
      21: areturn
```

在这个例子，通过 integer 和 double 的装箱很清楚地看到缺少了专门化。如果在处理应用的性能敏感区域时发现了大于等于 3 个元素的元组，你应该考虑定义一个不使用泛型的 case class 来避免装箱开销。这让 JVM 能够用基础类型而不是堆空间分配的对象来替代元组成员。

即使是使用 Tuple2，仍然可能产生装箱成本。

考虑如下代码：

```
case class Bar(value: Int) extends AnyVal
def tuple2Boxed: (Int, Bar) = (1, Bar(2))
```

基于我们已经掌握了的元组 2 和值类的字节码表示，我们期望这个方法产生两个在栈里分配的整数。不幸的是，在这个场景下，生成的字节码是这样的：

```
  public scala.Tuple2<java.lang.Object,
highperfscala.patternmatch.PatternMatching$Bar> tuple2Boxed();
    Code:
      0: new           #18  // class scala/Tuple2
      3: dup
      4: iconst_1
      5: invokestatic  #24  // Method
scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
      8: new           #26  // class
highperfscala.patternmatch/PatternMatching$Bar
     11: dup
     12: iconst_2
     13: invokespecial #29  // Method
highperfscala.patternmatch/PatternMatching$Bar."<init>":(I)V
     16: invokespecial #32  // Method
scala/Tuple2."<init>":(Ljava/lang/Object;Ljava/lang/Object;)V
     19: areturn
```

在前面的字节码里，我们看到 integer 被装箱，Bar 的实例被实例化。这个例子类似于

最后一个关于专门化的例子，就是包含 Container2 的那个。回看那个例子，显然 Container2 非常类似于 Tuple2。和前面一样，由于专门化的实现机制，在这种场景下编译器无法避免装箱。如果你在处理性能敏感代码，绕过的方法仍然是定义 case class。下面的例子证明定义 case class 消除了不期望的值类实例化和基础类型装箱操作：

```
case class IntBar(i: Int, b: Bar)
 def intBar: IntBar = IntBar(1, Bar(2))
```

这个定义生成了如下字节码：

```
  public highperfscala.patternmatch.PatternMatching$IntBar intBar();
    Code:
       0: new           #18  // class
highperfscala.patternmatch/PatternMatching$IntBar
       3: dup
       4: iconst_1
       5: iconst_2
       6: invokespecial #21  // Method
highperfscala.patternmatch/PatternMatching$IntBar."\<init>":(II)V
       9: areturn
```



注意 IntBar 没有定义为值类，因为它有两个参数。与元组版不同，这种写法既没有装箱也没有对 Bar 值类的引用。在这种场景下，定义一个 case class 带来了性能提升。

模式匹配

对于 Scala 新手，模式匹配常常是最容易掌握的语言特性之一，同时还打开了写代码的新思路。这种强大的机制使你能够用优雅的语法对不同的类型以编译器安全的方式做匹配。由于这种技术对于在 Scala 里写函数式范式的代码是如此重要，理解它的运行时开销是非常重要的。

字节码表示

考虑一个涉及到对代数数据类型做指令处理的例子。以下代码代表指令的一方：

```
sealed trait Side
case object Buy extends Side
case object Sell extends Side
def handleOrder(s: Side): Boolean = s match {
  case Buy => true
  case Sell => false
}
```



术语代数数据类型（ADT）是 sealed trait 及其 case 的正规称呼。比如说，Side、Buy 和 Sell 共同构造了一个 ADT。我们为实现业务定义了一组封装的 case。对于 Side，其封装的 case 是 Buy 和 Sell。sealed 修饰符的语义是封闭，因为用了这个语法就意味着不允许在别的源代码文件里扩展 Side。封闭的 ADT 语义使编译器能够推断模式匹配是否穷尽。如果你有意多学几个 ADT 的例子，可以参见第 2 章里的指令簿命令定义。

从下面的字节码可以看出 ADT 被去糖成一组 if 语句：

```
public boolean
handleOrder(highperfscala.patternmatch.PatternMatching$Side);
    Code:
       0: aload_1
       1: astore_2
       2: getstatic     #148  // Field
   highperfscala.patternmatch/PatternMatching$Buy$.MODULE$:Lhighperfscala.p
atternmatch/PatternMatching$Buy$;
       5: aload_2
       6: invokevirtual #152  // Method
   java/lang/Object.equals:(Ljava/lang/Object;)Z
       9: ifeq          17
      12: iconst_1
      13: istore_3
      14: goto          29
      17: getstatic     #157  // Field
   highperfscala.patternmatch/PatternMatching$Sell$.MODULE$:Lhighperfscala.
patternmatch/PatternMatching$Sell$;
```

```
      20: aload_2
      21: invokevirtual #152  // Method
java/lang/Object.equals:(Ljava/lang/Object;)Z
      24: ifeq          31
      27: iconst_0
      28: istore_3
      29: iload_3
      30: ireturn
      31: new           #159  // class scala/MatchError
      34: dup
      35: aload_2
      36: invokespecial #160  // Method
scala/MatchError."<init>":(Ljava/lang/Object;)V
```

通过查看字节码可以看到 Scala 编译器是符合用第 9 和第 24 下标的指令把模式匹配去糖成一组高效的 if 语句。这个示例展现了 Scala 的能力，它能够在提供表达力强且优雅的一等（first-class）语言特性的同时保持高效的字节码产出。

性能考虑

对持有状态的值（比如 case class）进行模式匹配会造成额外的运行时开销，光看 Scala 源代码不太容易看出这一点。下面这段代码扩展了前面的例子，引入了一些状态。

```
sealed trait Order
case class BuyOrder(price: Double) extends Order
case class SellOrder(price: Double) extends Order
def handleOrder(o: Order): Boolean = o match {
  case BuyOrder(price) if price > 2.0 => true
  case BuyOrder(_) => false
  case SellOrder(_) => false
}
```

这个例子比之前复杂了一些，因为除了要识别 3 个 case 的类型实例外，在第 1 个 case 里还对 BuyOrder 价格加了个断言。下面是用 scalac 移除了 scala 专有特性后的输出：

```
      case10(){
        if
(x1.$isInstanceOf[highperfscala.patternmatch.PatternMatching$BuyOrder]())
```

```
        {
          rc8 = true;
          x2 =
(x1.$asInstanceOf[highperfscala.patternmatch.PatternMatching$BuyOrder]():
highperfscala.patternmatch.PatternMatching$BuyOrder);
          {
            val price: Double = x2.price();
            if (price.>(2.0))
              matchEnd9(true)
            else
              case11()
          }
        }
      else
        case11()
    };
    case11(){
      if (rc8)
        matchEnd9(false)
      else
        case12()
    };
```

这个去糖后的代码展示了关于 Scala 编译器的几个有意思的问题。为了识别 `Order` 的类型而使用了 `java.lang.Object` 的 `isInstanceOf` 方法，对应于 `instanceOf` 字节码指令。通过 `asInstanceOf` 强制把 `Order` 转换成 `BuyOrder` 或 `SellOrder`。我们学到的第一点是对带有状态的类型做模式匹配会带来类型检查和强制类型转换的运行时成本。

第二个发现是对于第二个模式匹配 Scala 编译器能够通过创建一个叫作 `rc8` 的变量来判断 `BuyOrder` 是否已经被识别过了从而优化掉实例检查。如果自己是手写代码实现这个帅气的优化是不难，但是就破坏了模式匹配的优雅和简洁。这又是一个实例让我们看到 Scala 编译器能够从富有表达力的高阶代码生成高效的字节码。

通过前面的例子，已经可以确定模式匹配编译成 if 语句。对于关键路径代码，一个模式匹配的顺序是一个要考虑的性能因素。如果你的代码有 5 个模式匹配语句，其中第 5 个访问最频繁，那你的代码就在为前面 4 个分支的计算付出成本。我们来设计一个 JMH 微基准测试来评估模式匹配的线性访问成本。每个基准测试用不同值类型定义 10 个模式匹配

（比如值类、整数常量、case class，等等）。对每个基准测试扫一遍匹配下标以便展现访问第 1、第 5 和第 10 个匹配语句的成本。下面是微基准定义：

```
class PatternMatchingBenchmarks {
  @Benchmark
  def matchIntLiterals(i: PatternMatchState): Int = i.matchIndex match {
    case 1 => 1
    case 2 => 2
    case 3 => 3
    case 4 => 4
    case 5 => 5
    case 6 => 6
    case 7 => 7
    case 8 => 8
    case 9 => 9
    case 10 => 10
  }

  @Benchmark
  def matchIntVariables(ii: PatternMatchState): Int = ii.matchIndex match {
    case `a` => 1
    case `b` => 2
    case `c` => 3
    case `d` => 4
    case `e` => 5
    case `f` => 6
    case `g` => 7
    case `h` => 8
    case `i` => 9
    case `j` => 10
  }

  @Benchmark
  def matchAnyVal(i: PatternMatchState): Int = CheapFoo(i.matchIndex) match
{

    case CheapFoo(1) => 1
```

```
    case CheapFoo(2) => 2
    case CheapFoo(3) => 3
    case CheapFoo(4) => 4
    case CheapFoo(5) => 5
    case CheapFoo(6) => 6
    case CheapFoo(7) => 7
    case CheapFoo(8) => 8
    case CheapFoo(9) => 9
    case CheapFoo(10) => 10
  }

  @Benchmark
  def matchCaseClass(i: PatternMatchState): Int =
    ExpensiveFoo(i.matchIndex) match {
      case ExpensiveFoo(1) => 1
      case ExpensiveFoo(2) => 2
      case ExpensiveFoo(3) => 3
      case ExpensiveFoo(4) => 4
      case ExpensiveFoo(5) => 5
      case ExpensiveFoo(6) => 6
      case ExpensiveFoo(7) => 7
      case ExpensiveFoo(8) => 8
      case ExpensiveFoo(9) => 9
      case ExpensiveFoo(10) => 10
    }
}

object PatternMatchingBenchmarks {
  case class CheapFoo(value: Int) extends AnyVal
  case class ExpensiveFoo(value: Int)
  private val (a, b, c, d, e, f, g, h, i, j) = (1, 2, 3, 4, 5, 6, 7, 8, 9,
10)
  @State(Scope.Benchmark)
  class PatternMatchState {
    @Param(Array("1", "5", "10"))
    var matchIndex: Int = 0
  }
```

```
}
```

跑 30 轮测试，每轮持续 10 秒，加上 3 轮预热，每轮持续 5 秒。下面是基准测试结果：

```
sbt 'project chapter3' 'jmh:run PatternMatchingBenchmarks -foe true'
```

结果见下表：

基准测试	匹配的顺序	吞吐量（ops）	错误率（占吞吐量百分比）	变化的吞吐量百分比
matchAnyVal	1	350,568,900.12	±3.02	0
matchAnyVal	5	291,126,287.45	±2.63	-17
matchAnyVal	10	238,326,567.59	±2.95	-32
matchCaseClass	1	356,567,498.69	±3.66	0
matchCaseClass	5	287,597,483.22	±3.50	-19
matchCaseClass	10	234,989,504.60	±2.60	-34
matchIntLiterals	1	304,242,630.15	±2.95	0
matchIntLiterals	5	314,588,776.07	±3.70	3
matchIntLiterals	10	285,227,574.79	±4.33	-6
matchIntVariables	1	332,377,617.36	±3.28	0
matchIntVariables	5	263,835,356.53	±6.53	-21
matchIntVariables	10	170,460,049.63	±4.20	-49

最后一列用每个基准测试的匹配第一个分支的第一轮作为基础用例。对匹配第 5 和第 10 个分支的测试，在这一列显示性能下降的量。在所有场景下，除了匹配整数常量的场景外，吞吐量随着分支标号的变大而线性下降。唯一不符合规律的是匹配整数常量的场景。在这种场景下访问第 5 分支标号的测试性能反而相对提高了。通过检查字节码，我们发现在这种场景下编译器生成的是跳转表(jump table)而不是一组 if 语句。下面是字节码的片段：

```
6: tableswitch   { // 1 to 10
             1: 113
             2: 109
             3: 105
             4: 101
             5: 97
             6: 92
             7: 87
```

```
            8: 82
            9: 77
           10: 72
      default: 60
  }
```

这段字节码片段显示JVM把对整数常量的模式匹配用tableswitch指令转换成了跳转表。这是一个常量时间操作而不像遍历 if 分支那样是线性的。再看观察到的错误的百分比和观察到差异的百分比都只有几个百分点，我们可以得出结论线性成本不适用于这种场景。相反，由于生成了跳转表，对整数常量匹配来说访问第 N 个分支的访问成本是常量时间。与之相反，对整数变量的第 10 个分支做匹配的成本高大约两倍。从这个例子明显可以看出，在生成一系列 if 语句的模式匹配场景下，访问第 N 个分支的成本是线性的。如果在性能敏感代码里的模式匹配至少 3 个分支，那么建议复查一下代码看看分支顺序是否和访问频率相匹配。



你是否有只有两个分支的模式匹配？在直接匹配值且仅有两个分支的场景下，编译器能够生成高效的跳转表。当匹配基础类型常量（比如字符串常量或整数常量）时，编译器能够为大量的分支生成跳转表。类似于@tailrec 注解，Scala 定义了@switch 注解来让你可以告诉编译器你期望它生成跳转表。如果编译器无法生成跳转表而只能生成 if 语句，那编译器会给出警告。和@tailrec 注解类似，不管你是否加了@switch 注解，编译器都会启发式地应用跳转表。在实践中我们不常使用这个注解因为作用有限，但还是值得知道它的存在。下面这段加了注解的代码会编译成跳转表：

```
def processShareCount(sc: ShareCount): Boolean =
(sc: @switch) match {
case ShareCount(1) => true
case _ => false
}
```

尾递归

一个函数调用自身称为递归。递归是一种强大的工具，经常用于函数式编程。它使你

可以把复杂的问题分解成较小的子问题，使推理和解决变得容易。递归也和不变性（immutability）配合默契。递归函数给我们提供了一种很好的方法来管理改变的状态而不用使用可变数据结构或重新赋值变量。在本节里，我们专注于在 JVM 使用递归的几种缺陷，尤其是 Scala 里。

我们来看一个简单的递归方法例子。下面代码片段显示用于对整数类别求和的 sum 方法：

```
def sum(l: List[Int]): Int = l match {
  case Nil => 0
  case x :: xs => x + sum(xs)
}
```

前面代码里的 sum 方法叫作头递归（head-recursion）。sum(xs) 递归调动不是函数的最后一条指令。这个方法需要递归的结果来计算它自己的结果。考虑下面的调用：

```
sum(List(1,2,3,4,5))
```

运行时展开如下：

```
1 + (sum(List(2,3,4,5)))
1 + (2 + (sum(List(3,4,5))))
1 + (2 + (3 + (sum(List(4,5)))))
1 + (2 + (3 + (4 + (sum(List(5))))))
1 + (2 + (3 + (4 + (5))))
1 + (2 + (3 + (9)))
1 + (2 + (12))
1 + (14)
15
```

注意每次执行递归调用，我们的函数都挂起，等待右边的计算先完成好取得返回值。由于发起调用的函数只能在收到递归调用的结果后才能完成它自身的计算，每次调用都给调用栈增加一个新的入口。栈的大小是有限的，而我们可以对非常长的列表调用 sum。如果列表够长，调用 sum 可能导致 StackOverflowError：

```
$ sbt 'project chapter3' console
scala> highperfscala.tailrec.TailRecursion.sum((1 to 1000000).toList)
java.lang.StackOverflowError
```

```
    at scala.collection.immutable.Nil$.equals(List.scala:424)
    at highperfscala.tailrec.TailRecursion$.sum(TailRecursion.scala:12)
    at highperfscala.tailrec.TailRecursion$.sum(TailRecursion.scala:13)
    at highperfscala.tailrec.TailRecursion$.sum(TailRecursion.scala:13)
    at highperfscala.tailrec.TailRecursion$.sum(TailRecursion.scala:13)
...omitted for brevity
```

堆栈跟踪显示所有递归调用都堆在栈上，等待下一步的结果。这证明了没有一个调用能够在完成递归调用前完成自己的计算。我们的栈空间在完成最后一个调用前就爆掉了。

为了避免这个问题，我们需要把我们的方法重构为尾递归（tail-recursive）。如果一个递归方法里的最后一条指令是递归调用则称为尾递归。尾递归方法可被优化为类似 while 循环而不是一系列递归调用。这意味着只有第一个调动被压到栈里：

```
def tailrecSum(l: List[Int]): Int = {
  def loop(list: List[Int], acc: Int): Int = list match {
    case Nil => acc
    case x :: xs => loop(xs, acc + x)
  }
  loop(l, 0)
}
```

这个新版的 sum 是尾递归的。注意我们构造了一个内部的 loop 方法，它接受待求和的列表和一个累加器来计算结果的当前状态。loop 方法之所以是尾递归是因为递归调用 loop(xs,acc+x) 是最后一条指令。通过每次循环的时候计算累加器，我们避免了把递归调用压栈：

```
scala> highperfscala.tailrec.TailRecursion.tailrecSum((1 to
1000000).toList)
res0: Int = 1784293664
```



我们提过，递归是函数式编程的重要方面。但是在实践中，我们应该很少需要亲手写递归方法，尤其是在处理类似 List 这样的集合时。标准库 API 提供了已经优化好的方法，我们应该优先选用。比如说整数列表求和可以写成这样：

```
list.foldLeft(0)((acc, x) => acc + x)
```

```
list.foldLeft(0)(+)
```

foldLeft 函数内部使用 while 循环实现的，不会造成 StackOverflowError 异常。

实际上，List 本身就有 sum 方法，使整数列表求和更容易了。sum 方法是用类似上面的 foldLeft 实现的。

字节码表示

事实上，JVM 不支持尾递归优化。为了达到目的，Scala 编译器在编译期把尾递归方法优化成 while 循环。我们比较两种实现下的字节码。

我们原本的头递归的 sum 方法编译成如下的字节码：

```
public int sum(scala.collection.immutable.List<java.lang.Object>);
    Code:
       0: aload_1
// omitted for brevity
      52: invokevirtual #41  // Method
sum:(Lscala/collection/immutable/List;)I
      55: iadd
      56: istore_3
      57: iload_3
      58: ireturn
// omitted for brevity
```

而尾递归的 loop 方法生成如下字节码：

```
    private int loop(scala.collection.immutable.List<java.lang.Object>,
int);
      Code:
         0: aload_1
       // omitted for brevity
        60: goto          0
      // omitted for brevity
```

注意指令 52 用 invokevirtual 指令调用自身，然后还得再对返回值执行一些指令。与之相反，loop 方法在指令 60 使用 goto 指令调回代码块头上，这样就避免了把对自身

调用压栈。

性能考虑

编译器只能优化简单的尾递归场景，也就是最后一句指令是递归调用指令的递归方法。有很多边缘场景都可以描述为尾递归，但是太复杂以至于编译器无法优化。为了避免无意中写了不可优化的尾递归方法，你应该始终给尾递归方法加上@tailrec。@tailrec 注解告诉编译器，“我相信你应该能优化这个递归方法；如果优化不了，请在编译时给我报错”。要记住的是@tailrec 并非告诉编译器去优化方法，不管你加不加编译器都会去优化。注解用于让开发者确定编译器能够优化递归。



这时候你应该已经注意到所有的 while 循环都可以用尾递归来替代而不损失性能。如果你在 Scala 里用了 while 循环，你可以想一想怎么把它们替换成尾递归实现。尾递归可以消除可变变量。

下面是加了@tailrec 注解的 tailrecSum 方法：

```
def tailrecSum(l: List[Int]): Int = {
  @tailrec
  def loop(list: List[Int], acc: Int): Int = list match {
    case Nil => acc
    case x :: xs => loop(xs, acc + x)
  }
  loop(l, 0)
}
```

如果我们尝试在第一个头递归实现上加注解，编译器会报错：

```
[error]
chapter3/src/main/scala/highperfscala/tailrec/TailRecursion.scala:12: could
not optimize @tailrec annotated method sum: it contains a recursive call
not in tail position
[error] def sum(l: List[Int]): Int = l match {
[error]                              ^
[error] one error found
[error] (chapter3/compile:compileIncremental) Compilation failed
```

我们推荐始终使用@tailrec 来确保你的方法能被编译器优化。由于编译器只能优化比较简单的尾递归，因此在编译期确保你不会无意中写了不可优化的函数而造成 StackOverflowError 异常是很重要的。我们现在来看几个编译器优化不了的案例：

```
def sum2(l: List[Int]): Int = {
 def loop(list: List[Int], acc: Int): Int = list match {
   case Nil => acc
   case x :: xs => info(xs, acc + x)
 }
 def info(list: List[Int], acc: Int): Int = {
   println(s"${list.size} elements to examine. sum so far: $acc")
   loop(list, acc)
 }
 loop(l, 0)
}
```

sum2 的 loop 方法不能被优化，因为它有两个不同的方法互相调用。如果我们把对 info 的调用替换成实现，就可以优化了：

```
def tailrecSum2(l: List[Int]): Int = {
 @tailrec
 def loop(list: List[Int], acc: Int): Int = list match {
   case Nil => acc
   case x :: xs =>
    println(s"${list.size} elements to examine. sum so far: $acc")
    loop(list, acc)
 }
 loop(l, 0)
}
```

一个有点类似的场景是编译器无法处理 by-name 参数：

```
def sumFromReader(br: BufferedReader): Int = {
 def read(acc: Int, reader: BufferedReader): Int = {
   Option(reader.readLine().toInt)
     .fold(acc)(i => read(acc + i, reader))
 }
 read(0, br)
```

```
}
```

read 方法无法优化，因为它无法通过 Option.fold 的定义理解到这实际上是尾递归。如果我们把 fold 调用替换成它的实际实现，我们就可以加上注解：

```
def tailrecSumFromReader(br: BufferedReader): Int = {
  @tailrec
  def read(acc: Int, reader: BufferedReader): Int = {
    val opt = Option(reader.readLine().toInt)
    if (opt.isEmpty) acc else read(acc + opt.get, reader)
  }
  read(0, br)
}
```

编译器还会拒绝优化非 public 方法。这是为了防止子类把父类的方法覆盖成非尾递归的版本。超类里的尾递归调用可能会转调到子类的实现上导致破坏尾递归：

```
class Printer(msg: String) {
 def printMessageNTimes(n: Int): Unit = {
   if(n > 0){
     println(msg)
     printMessageNTimes(n - 1)
   }
 }
}
```

试图给 printMessageNTimes 标注为尾递归会产生如下错误：

```
[error]
chapter3/src/main/scala/highperfscala/tailrec/TailRecursion.scala:74: could
not optimize @tailrec annotated method printMessageNTimes: it is neither
private nor final so can be overridden
[error] def printMessageNTimes(n: Int): Unit = {
[error]     ^
[error] one error found
[error] (chapter3/compile:compileIncremental) Compilation failed
```

另一个无法优化的递归场景是在这个递归属于某个 try/catch 块的时候：

```
def tryCatchBlock(l: List[Int]): Int = {
 def loop(list: List[Int], acc: Int): Int = list match {
   case Nil => acc
   case x :: xs =>
     try {
       loop(xs, acc + x)
     } catch {
       case e: IOException =>
         println(s"Recursion got interrupted by exception")
         acc
     }
 }
 loop(l, 0)
}
```

和前面的例子相反，在这个例子里编译器没有抱怨。递归调用不在尾递归位置上。由于它被包在 try/catch 块里，方法需要准备接受一个潜在的异常然后执行某些操作来处理。为了证明这点，我们可以看看生成的字节码，可以看到最后的指令是关于 try/catch 的：

```
private final int loop$4(scala.collection.immutable.List, int);
   Code:
      0: aload_1
     // omitted for brevity
     61: new           #43  // class scala/MatchError
     64: dup
     65: aload_3
     66: invokespecial #46  // Method
scala/MatchError."<init>":(Ljava/lang/Object;)V
     69: athrow
     // omitted for brevity
     114: ireturn
   Exception table:
      from    to  target type
         48    61    70   Class java/io/IOException
```

希望这几个例子可以让你看到很容易写出非尾递归的方法。防止这种错误的最好方法就是始终加上@tailrec 注解来检验。

Option 数据类型

Option 数据类型在 Scala 标准库里广为使用。和模式匹配一样，这也是 Scala 初学者很早就会采用的语言特性。Option 数据类型提供了一种优雅的方法来转换和处理不是一定存在的值，废除了 Java 代码里常常能见到的 null 检查。我们假设你已经理解而且欣赏 Option 给 scala 函数式编程带来的价值，所以就不在重复讲解它的好处了。我们专注于分析它的字节码表示来掌握它的性能特点。

字节码表示

通过查看 Scala 源代码，我们看到 Option 实现为一个抽象类，有两种实现，Some 和 None。下面展示了类型声明（去掉了实现代码以方便阅读）：

```
sealed abstract class Option[+A] extends Product with Serializable
final case class Some[+A](x: A) extends Option[A]
case object None extends Option[Nothing]
```

通过研究类定义，我们可以对字节码表示做几个推断。先专注于 Some，我们注意到它没有扩展 AnyVal。因为 Option 是用继承实现的，由于我们在值类那节讲解过的限制，Some 无法设计成值类。这个限制意味着每个包装在 Some 实例里的值都需要分配内存。我们更进一步观察到 Some 没有专门化。根据我们之前学习的专门化知识，我们注意到包装在 Some 实例里的基础类型需要装箱。下面这个简单的例子同时展示了这两点：

```
def optionalInt(i: Int): Option[Int] = Some(i)
```

在这个小例子里，我们把一个整数包装在 Some 实例里以便用作 Option 数据类型。生成的字节码如下：

```
  public scala.Option<java.lang.Object> optionalInt(int);
    Code:
       0: new           #16  // class scala/Some
       3: dup
       4: iload_1
       5: invokestatic  #22  // Method
scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
       8: invokespecial #25  // Method
scala/Some."<init>":(Ljava/lang/Object;)V
```

```
    11: areturn
```

正如我们预期的，分配了一个对象来创建 Some 实例，紧接着把传入的整数装箱来构造 Some 实例。

None 实例从字节码的角度来看简单很多。由于 None 定义为 Scala 对象，所以没有创建 None 实例的实例化成本。这是合理的，因为 None 代表没有状态需要维持的场景。



你有没有注意到单一的None值怎么能代表各种类型的值？答案在于 None 是 Nothing 类型。Nothing 类型扩展所有的其他类型，这样就允许 None 是任意 A 类型的子类。要了解更多关于 Scala 类型层次的知识，请查看 Scala 语言教程 http://docs.scala-lang.org/tutor ials/tour/unified-types.html。

性能考虑

在任何性能不敏感的环境下，应该默认用 Option 来代表可能不存在的值。在性能敏感的领域，这个选择就变得不那么清晰了。尤其对于性能敏感的代码，我们首先要为正确性优化，其次再为性能优化。我们建议第一个版本永远先用符合编程管理的方式实现，也就是说，用 Option。根据对 Some 的字节码表示的理解，符合逻辑的下一步是通过 profile 来判断 Option 是否是瓶颈。尤其是当你专注于内存分配模式和垃圾收集成本时。根据我们的经验，一般有其他的瓶颈要比 Option 的成本更高。包括低效的算法实现，构建的很差的领域模型，或者低效的系统资源利用等。如果在你的场景下，别的低效原因都已经排除，而 Option 是最后的性能问题时，你就需要采取进一步的行动了。

逐步提高性能的步骤可能包括停止使用 Option 的高阶函数。在关键路径上，把高阶函数替换成内联的等价实现可能带来可观的成本节约。考虑下面这个把 Option 数据类型转换成 String 数据类型的小例子：

```
Option(10).fold("no value")(i => s"value is $i")
```

在关键路径上，下面这种修改可能带来可观的提升：

```
val o = Option(10)
if (o.isDefined) s"value is ${o.get} else "no value"
```

用 if 语句替换 fold 操作节约了创建匿名函数的成本。再重复一次，只有在经过详细的 profile 证明 Option 是瓶颈的情况下才考虑这种优化。尽管这种代码修改有可能提高性能，但是它既烦琐又由于使用了 o.get 而变得不安全。如果滥用这种技巧，你还不如在关键路径上保持使用 Option。

如果把高阶函数替换成内联函数和不安全的等价实现也不能有效地提高性能，那你可能需要考虑更激进的度量。这时候 profile 应该揭示 Option 内存分配是瓶颈，使你无法达成性能目标。面对这种场景，你有两个选择，两种选择实现起来时间成本都很高。

一种是承认对于关键路径不适用 Option，必须全部去除替换成 null 检查。这是性能最好的步骤，但是带来显著的维护成本增加，因为你和所有在关键路径上工作的团队成员都必须认知这个决策。如果决定采用这个策略，你需要定义明确的边界来把关键路径隔离在最小的范围里。在下一节里，我们谈论第二种方法，构造一种新的数据类型发挥我们之前学到的知识。

案例研究——性能更高的 Option

如果你还不打算丢失 Option 数据类型包含的信息（使用 null），那么你也许希望探索另一种方法来实现一个垃圾回收更友好的 Option 实现。本节里，我们展示了另一种实现方法，在同样保证类型安全的情况下避免装箱和 Some 实例实例化。我们利用标记类型和专门化而且不允许 null 作为 Some 的合法值，做出了一种实现，如下：

```
sealed trait Opt
object OptOps {
  def some[@specialized A](x: A): A @@ Opt = Tag(x)
  def nullCheckingSome[@specialized A](x: A): A @@ Opt =
    if (x == null) sys.error("Null values disallowed") else Tag(x)
  def none[A]: A @@ Opt = Tag(null.asInstanceOf[A])
  def isSome[A](o: A @@ Opt): Boolean = o != null
  def isEmpty[A](o: A @@ Opt): Boolean = !isSome(o)
  def unsafeGet[A](o: A @@ Opt): A =
    if (isSome(o)) o.asInstanceOf[A] else sys.error("Cannot get None")
  def fold[A, B](o: A @@ Opt)(ifEmpty: => B)(f: A => B): B =
    if (o == null) ifEmpty else f(o.asInstanceOf[A])
}
```

这个实现定义了工厂方法类构造可选的类型（也就是，some、nullCheckingSome 和 none）。和 Scala 标准 Option 不同，这个实现用标记类型来给值添加类型信息而不是创建一个新值再封装。none 的实现利用 null 在 scala 里是一个值而不是像 Java 里面是个关键字这个优势。记住如果不是性能需求必须要采取这种极端措施，我们不会采用这种秘传招数。每个工厂方法返回的标记类型保持了类型安全，需要显式的解包来访问里面的类型。



如果你想了解更多关于 Scala 如何表示 null 值，我们推荐你查看这两个 StackOverflow 帖子：http://stackoverflow. com/questions/8285916/why-doesnt-null-asinstanceofint-throw-a-nullpointerexception http://sta ckoverflow.com/questions/10749010/if-an- int-cant-be-null-what-does-null-asinstanceofint-mean。在这个两个帖子里，多名回复者提供了非常好的回复帮助你加深理解。

OptOps 里其余的方法定义了你可以在 Scala Option 里找到的方法。与实例方法不同，我们可以看见这些方法是静态的，因为工厂方法不再分配新实例了。理论上可以定义一个隐式类来提供类似实例方法的语法模拟效果，但我们不会这么做，因为我们现在是假设正处于极端的性能要求下。语义上说，OptOps 里的操作等价于 Scala Option 里的对应操作。我们不用把一个值和无值（也就是 None）去匹配，我们可以利用把 null 当作值来引用的优势。

使用这个实现，运行时开销包括实例检查和调用 scalaz.Tag。我们失去了模式匹配的能力而必须用 fold，或者在极端情况下用 isSome 和 unsafeGet。为了更好地理解运行时差异，我们对标准库的 Option 和我们刚才实现的 Option 做了微基准测试。你可以通过微基准代码体会一下语法上的不同。我们鼓励你运行 javap 来反汇编字节码来证明这个实现确实避免了装箱和对象创建：

```
class OptionCreationBenchmarks {

  @Benchmark
  def scalaSome(): Option[ShareCount] = Some(ShareCount(1))

  @Benchmark
  def scalaNone(): Option[ShareCount] = None
```

```
  @Benchmark
  def optSome(): ShareCount @@ Opt = OptOps.some(ShareCount(1))

  @Benchmark
  def optSomeWithNullChecking(): ShareCount @@ Opt =
    OptOps.nullCheckingSome(ShareCount(1))

  @Benchmark
  def optNone(): ShareCount @@ Opt = OptOps.none

  @Benchmark
  def optNoneReuse(): ShareCount @@ Opt = noShares
}

object OptionCreationBenchmarks {
  case class ShareCount(value: Long) extends AnyVal
  val noShares: ShareCount @@ Opt = OptOps.none
}
```

下面的命令大家应该很熟悉了：

```
sbt 'project chapter3' 'jmh:run OptionCreationBenchmarks -foe true'
```

结果总结在下表里：

基准测试	吞吐量（ops）	错误率（占吞吐量百分比）
optNone	351,536,523.84	±0.75
optNoneReuse	344,201,145.90	±0.23
optSome	232,684,849.83	±0.37
optSomeWithNullChecking	233,432,224.39	±0.28
scalaNone	345,826,731.05	±0.35
scalaSome	133,583,718.28	±0.24

可能令人印象最深刻的是在使用标记类型的版本时比 Scala 标准库实现的版本的吞吐量提高了大约 57%。

这可能是由于减少了内存分配压力。我们看到 None 创建的吞吐量是类似的。我们还观察到在构造标记的 Some 选项时加入 null 检查似乎没有成本。如果你信任的团队不会

乱传 null 值，那么这个检查是多余的。我们还构造了一组基准测试来评估 fold 的性能，以感受一下这种 Option 变种实现的相对成本。下面是简单的 fold 基准测试的代码：

```
class OptionFoldingBenchmarks {
  @Benchmark
  def scalaOption(): ShareCount =
    scalaSome.fold(ShareCount(0))(c => ShareCount(c.value * 2))
  @Benchmark
  def optOption(): ShareCount =
    OptOps.fold(optSome)(ShareCount(0))(c => ShareCount(c.value * 2))
}
object OptionFoldingBenchmarks {
  case class ShareCount(value: Long) extends AnyVal
  val scalaSome: Option[ShareCount] = Some(ShareCount(7))
  val optSome: ShareCount @@ Opt = OptOps.some(ShareCount(7))
}
```

用跟前面一样的参数运行：

```
jmh:run OptionFoldingBenchmarks -foe true
```

结果总结在下表里：

基本测试	吞吐量（ops）	错误率（占吞吐量百分比）
optOption	346,208,759.51	±1.07
scalaOption	306,325,098.74	±0.41

在这个基准测试里，我们希望证明在用标记类型实现的 Option 变种和 Scala 标准 Option 的吞吐量没有显著下降。如果这里的性能显著下降，那我们在创建实例基准测试里获取的提升可能就被抵消了。幸运的是，基准测试表明 fold 吞吐量实际上提高了大约 13%。



看到基准测试得出了验证猜想的结果让人松了一口气。但是理解好的结果为什么出现，能够给出解释，是一样重要的。如果不理解这些结果是怎么出来的，你很可能无法重现结果。你怎么解释标记类型的实现比 Scala Option 实现吞吐量提升了？考虑一下我们的实现和前面讨论过的内存分配差异。

基准测试表明标记类型的 Option 实现比 Scala Option 实现在性能上有质的提升。如果你面对性能问题而 profile 揭示 Scala Option 是瓶颈，研究一下这个变种实现是值得考虑的。尽管性能提升了，但也有一些损失。当使用变种的实现时，你失去了模式匹配的能力。更大的代价是和标准库以及第三方库的一致性。如果你的关键路径存在和标准库或第三方库的大量交互，你将被迫重写大量的代码以便使用变种的 Option 实现。在这种情况下，如果时间压力很紧，你可能要重新考虑是否用 null 来为部分领域建模更有意义一些。如果你的关键路径代码很少和标准库或第三方库交互，那么使用变种 Option 实现可能是更简单的选择。

我们的案例分析基于 Alexandre Bertails 在他的博客文章里探索的新方法：https://bertails.org/2015/02/15/abstract-algebraic-data-type/。他用一种被他称为抽象代数数据类型的方法解决相同的性能问题。两种方法都基于用类型约束来为 Option 建模而避免实例分配。通过在 Option 代数数据类型之上进一步抽象，你设计了一种无须内存分配和装箱的实现。我们建议你探索这个方法，因为这是另一个在保持类型安全的同时提供高性能的好例子。

小结

在本章里，我们深入学习了常用的 Scala 语言特性的字节码表示及其性能考虑。在案例研究里，我们看到了如何把 Scala 语言特性和优秀的 Scalaz 库结合来构造一个更适用于高性能需要的 Option 实现。

在所有例子里，我们反复强调在考虑性能取舍时保证类型安全和正确性。作为函数式程序员，我们强调编译期正确性和引用透明性。甚至当我们在标记类型的 Option 实现里使用 null 时，仍然保证了正确性，因为 null 值是内部实现细节。当你反思我们讨论过的策略时，考虑我们是如何保持引用透明的（也就是无副作用）代码同时仍然能够达到性能目标。

此时，你应该对 Scala 优雅的语言特性在实现时对于性能的取舍更有信心了。通过我们的分析，你学到了如何把简洁的 Scala 语法翻译到 JVM 字节码。这在排查性能问题时是非常有价值的技能。随着你研究更多的例子，你会发展出哪里可能有潜在问题的更强的直觉。将来你可以回头参考本章来复查常用的补救策略来在性能、安全和优雅之间做取舍。下一章，我们将通过深入集合库来继续学习如何发挥 Scala 的能力来编写高性能的函数式编程代码。

4

探索集合 API

在这一章里，我们将回到 MV 交易公司的场景里来迎接来自各个项目团队的挑战。市场数据团队需要一个针对关键路径优化过的程序来处理日益增长的取消指令请求。数据科学团队想要一个更好的定制化数据分析工具来研究交易策略。每个人都还有一个昨天没有解决的问题。这些就是起步时我们的工作节奏！

接下来，将使用函数式编程范式，用我们已有的知识经验以及 Scala 集合 API 给我们带来的优势来解决这些问题。Scala 语言和它的集合 API 的强大功能让我们可以用也许以前从未想过的方式来处理问题。随着我们解决一个个的问题，接触一个个 Scala 集合的使用方法，我们会详细展示集合的实现以及需要进行折中考虑的地方。我们将会在这一章来考查以下这些集合 API：

- List
- TreeMap
- Queue
- Set
- Vector
- Array

高吞吐量系统 - 改进指令簿

在第 1 章里，你在一种紧张的情势下结识了 MV 交易公司的交易员主管 Dave。金融市场经历的一场剧变暴露了指令簿的设计缺陷。在和 Dave 沟通之后，你知道了在一个易变的

市场环境下，指令系统的吞吐量是受取消指令的数量所支配的，因为在快速调整的市场环境下，交易员们需要迅速做出反应。通过对指令簿进行的基准分析和性能分析，你确定了之前的怀疑：在高吞吐量情况下，处理取消指令的性能将影响系统延迟的程度。

虽然由于市场变化导致的商业损失已经过去，Dave 意识到了未来的市场变化对于 MV 交易公司的营收将仍然是一个风险。Dave 想要进行工程上的投资，使指令簿在取消指令频繁发生的情况下有更高的性能。在和数据科学团队一起研究之后，Dave 分析了指令簿系统里 3 个月以来发生的活动事件，并发现了一些有趣的特征。他和你分享了 3 个月以来基于交易日的数据分析结果，平均下来取消指令构成了 70%的交易请求。这项分析同时展示了在市场变化最快的数天里，取消指令的活动要占到整个系统活动的 85%。以俏皮闻名的 Dave 总结说：现在，我知道的你们都知道了，之后就看你们怎么执行了。

理解过去实现上的折中 - list 实现

提升指令簿的性能是一件激动人心的事情，你优化的第一步是熟悉指令簿的实现。打开指令簿的代码仓库之后，你联系了 Gary，一个具有之前系统开发经验的同事。Gary 知道指令簿开发的历史，他告诉你去检查 `ListOrderBook` 的代码。他说："这是我们建模指令簿的第一次尝试，你看了这个版本之后应该可以看出我们的设计思路。理解它的实现之后，接着可以看看 `QueueOrderBook` 的实现。那是指令簿的第二个版本。在经历市场波动的时候，你们曾对旧版本的实现进行过性能分析。有问题的话请随时找我。"在感谢过他之后，你钻研了代码仓库以查找 `ListOrderBook` 的代码。`ListOrderBook` 类定义了以下的状态来管理卖出和买入：

```
case class ListOrderBook(
    bids: TreeMap[Price, List[BuyLimitOrder]],
    offers: TreeMap[Price, List[SellLimitOrder]]) {
    def bestBid: Option[BuyLimitOrder] =
      ??? // 省略
    def bestOffer: Option[SellLimitOrder] =
      ??? // 省略
}
```

为了唤醒大家的记忆，让我们再来看看 `Price`、`BuyLimitOrder` 和 `SellLimitOrder` 的定义：

```
sealed trait LimitOrder {
```

```
    def id: OrderId
    def price: Price
}
case class BuyLimitOrder(id: OrderId, price: Price) extends LimitOrder
case class SellLimitOrder(id: OrderId, price: Price) extends LimitOrder
case class Price(value: BigDecimal)
```

LimitOrder 是一个表示两种限制指令的一个**代数数据结构**（algebraic data type，ADT）。Price 类是 BigDecimal 的一个增强的包装类型。回想起值类可以用于提高性能，基于此，你修改 Price 的定义如下：

```
case class Price(value: BigDecimal) extends AnyVal
```

ListOrderBook 类使用了两种 Scala 的集合类型，List 和 TreeMap，来维护其内部状态。接下来，为了理解系统设计上的折中，我们来深入研究一下这些数据结构。

List

Scala 将 List 实现为一个不可变的单链表。一个 List 是一个有序的同类型的元素集合，同时它是一个有两个实现的抽象类：代表空表的 Nil 和用于表示一个元素及尾列表的“::”。更具体地讲，让我们来看看一些跟实际实现类似的伪代码：

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class ::[A](head: A, tail: List[A]) extends List[A]
```

一个包含 3 个整数的 List 可以由下面的语句来构建：

```
val list = ::(1, ::(2, ::(3, Nil)))
```



注意在 List 特质定义里面的加号，这个加号（+）表示 List 是根据其类型参数 A 而协变的（convariant on its type parameter, A）。协变的能力使我们可以使用泛型来表达多态。更具体地讲，考虑以下定义：

```
sealed trait Base
case class Impl(value: Int) extends Base
```

这里的语句表达了 Base 和 Impl 之间的一种关系：Impl 是 Base 的子类型。当和 List 一起使用的时候，协变使我们将 List[Impl]

表达为 List[Base]的一个子类型。用实例来说明就是，协变允许以下代码可以编译：

```
val bases: List[Base] = List[Impl](Impl(1))
```

协变实际上属于更宽泛的关于变性主题的内容。如果你想学习更多关于Scala变性的知识，请参考Andreas Schroeder的这篇精彩博客：https://blog.codecentric.de/en/2015/03/scala-type-system-parameterized-types-variances-part- 1/。

和其他的大多数 Scala 集合不同的是，List 支持对其内容进行模式匹配。这种强大的功能，使我们可以编写富有表达力的代码：在处理了各种可能的场景的同时保持编译时类型安全。考虑以下的代码片段：

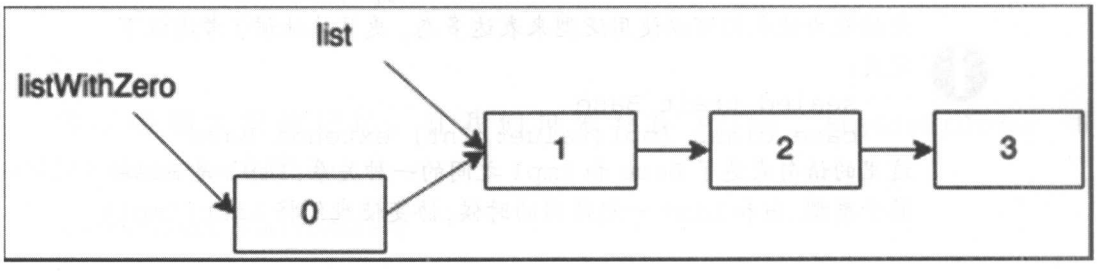
```
List(1,2,3,4) match {
    case 1 :: x :: rest => println(s"second element: $x, rest: $rest")
}
```

在这个简单的模式匹配中，我们成功地表达了对多个问题的处理。这个例子中，1 是 1，x 是 2，rest 是 List(3, 4)。在编译阶段，这个代码片段会引出一个编译警告，因为 Scala 编译器推断出可能有一些 List 模式没有被匹配到（例如空表）。编译器提供的警告可以最小化忘记处理有效输入的概率。

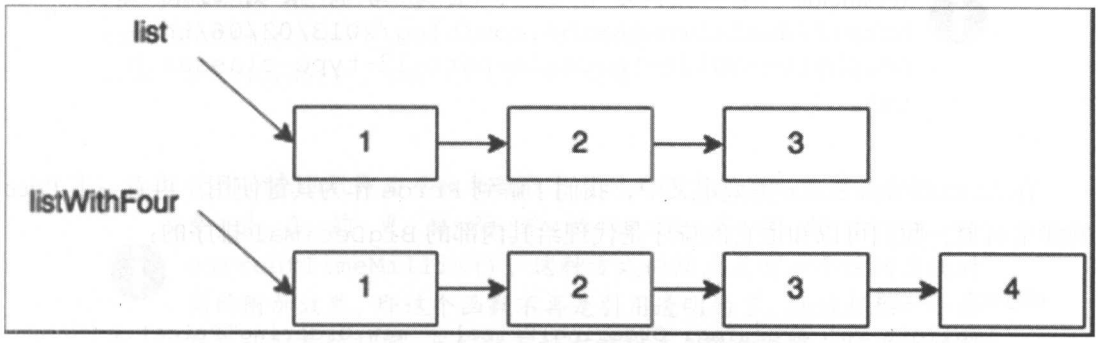
List 对于向前增加（prepend）元素的操作进行了优化。在已有列表里面增加元素 0 就像下面这样简单：

```
val list = ::(1, ::(2, ::(3, Nil)))
val listWithZero = ::(0, list)
```

由于 List 实现了数据共享，这使得它成为一个常数时间的操作，并且几乎没有内存消耗。换言之，新生成的列表 listWithZero 并不是一个 list 的深层拷贝，它重用了所有已经分配的元素，然后分配了一个新的包含 0 的元素：



对比向前增加元素的操作，向后增加元素（append）操作则是计算开销大的操作，因为整个 List 都必须被拷贝。



由于向后增加元素的列表操作的性能如此之低，你可能会考虑我们是否可以安全地使用 map 变换。map 变换对一个列表里面的元素连续地应用一个函数，它在逻辑上表现为向一个新的 List 里面向后增加变换后的值。为了避免性能陷阱，List.map 覆盖了特质 TraversableOnce 提供的默认实现，使用向前增加元素操作进行实现。这种实现方式在提升 List.map 性能的同时保持了 API 不变。覆盖默认行为提供特定实现是 Scala 集合库里面的一种通常的做法。常数时间的表头操作使 List 成为一些算法的理想工具，如后进先出（LIFO）算法。而对于随机访问和先进先出（FIFO）的行为，你应该有选择性地使用 List。

在接下来的一节里面，我们会研究 TreeMap。TreeMap 是一个 SortedMap 特质的实现，它被用于维护卖出和买入信息。

TreeMap

TreeMap 类是一个根据提供的排序策略对键进行排序的映射表。以下类定义片段清晰地显示了这个排序特性：

```
class TreeMap[A, +B] private (tree: RB.Tree[A, B])(implicit val ordering:
  Ordering[A])
```

Ordering 类是一个视界（类型类，type class），它为 A 类型元素的自然序定义了协议。



如果视界对于你而言是一个新概念，那我们推荐你去读一下 Daniel Westheide 的一篇关于这个主题写得很好的博客 http://danielwestheide.com/blog/2013/02/06/the-neophytes-guide-to-scala-part-12-type-classes.html。

在 ListOrderBook 类型定义中，我们了解到 Price 作为其键使用。再看一下 Price 的伴生对象，我们可以知道它的排序是代理给其内部的 BigDecimal 排序的：

```
object Price {
  implicit val ordering: Ordering[Price] = new Ordering[Price] {
    def compare(x: Price, y: Price): Int =
      Ordering.BigDecimal.compare(x.value, y.value)
  }
}
```

由 ListOrderBook 所引用的 TreeMap，如同 List 一样具有不变性。不变性给代码推理提供了一个强有力的保证。由于从一个 map 里面增加和删除元素总是会以一个新 map 对象的形式返回，我们可以确信这不会带来什么副作用。

TreeMap 实现为一个特殊的二叉搜索树，即红黑树。这种树实现方式对元素查找、增加、删除这些操作提供了对数的时间复杂度。在看到 TreeMap 而不是 HashMap 时，你可能会觉得奇怪。因为根据 Scala 集合性能概述文档（http://docs.scala-lang.org/overviews/collections/performance-characteristics.html）中的描述，HashMap 对元素查找、增加、删除操作提供常数时间复杂度，这比 TreeMap 更快。然而，TreeMap 在依序遍历元素时提供了优异的性能。例如，在 map 里面查找最大的键，TreeMap 可以以对数时间完成而 HashMap 需要线性的时间才能完成。这个区别暗示了指令簿的实现需要高效的按序对 Price 进行遍历。

增加 limit orders

回到 ListOrderBook 的实现，我们可以看到以下方法定义（一部分），它们可以反映指令簿的核心：

```
def handle(
```

```
    currentTime: () => EventInstant,
    ob: ListOrderBook,
    c: Command): (ListOrderBook, Event) = c match {
    case AddLimitOrder(_, o) => ??? // 省略
    case CancelOrder(_, id) => ??? // 省略
}
```



使用一个函数作为参数来获取当前时间的实现看起来有点古怪。一个潜在的更为简单的方法是调用 System.currentTimeMillis()。这种方式的缺点是有一个访问系统时间的附加效果，即这个函数不再是引用透明的了。通过提供一个函数来获取系统时间，我们就有能力控制这个附加效果如何发生并编写可重复的测试用例。

给定一个 Command，一个指令簿实例，一种获取事件发生时间的方式，一个事件和一个新的状态就可以生成出来。回忆一下下面指令簿可以处理的命令：

```
sealed trait Command
case class AddLimitOrder(i: CommandInstant, o: LimitOrder) extendsCommand
case class CancelOrder(i: CommandInstant, id: OrderId) extends Command
```

以下是可能的由处理命令创建的事件：

```
sealed trait Event
case class OrderExecuted(i: EventInstant, buy: Execution,
    sell: Execution) extends Event
case class LimitOrderAdded(i: EventInstant) extends Event
case class OrderCancelRejected(i: EventInstant,
    id: OrderId) extends Event
case class OrderCanceled(i: EventInstant,
    id: OrderId) extends Event
```

现在让我们将注意力集中在 AddLimitOrder 命令的实现上，以便更好地理解那些由历史设计选择导致的算法属性。当增加一个指令时，可能会发生以下两种情况之一：

- 新进的指令价格适合指令簿，产生 OrderExecuted。
- 新进的指令不适合指令簿，产生 LimitOrderAdded。

判断新进指令是否适合指令簿需要在指令簿的对端查找最佳价格。让我们再看看 LimitOrderBook 的定义及 bestBid 和 bestOffer 的完整实现，如下：

```
case class ListOrderBook(
    bids: TreeMap[Price, List[BuyLimitOrder]],
    offers: TreeMap[Price, List[SellLimitOrder]]) {
    def bestBid: Option[BuyLimitOrder] =
        bids.lastOption.flatMap(_._2.headOption)
    def bestOffer: Option[SellLimitOrder] =
        offers.headOption.flatMap(_._2.headOption)
}
```

在这个实现中，我们利用了 TreeMap 在有序搜索时能达到对数复杂度的优势。最佳买入就是买入价格最大值时的买入，由于 tree 是升序排序的，这个买入就是 tree 里面的最后一个元素。最佳出售就是最低出售价格的出售，也就是 tree 里面的第一个元素。

让我们特别关注一下增加一个买入指令和提供一个最佳出售指令，以下的比较操作将在决定新进的买入指令成交与否时发生：

```
orderBook.bestOffer.exists(buyOrder.price.value >= _.price.value)
    match {
        case true => ??? // cross the book
        case false => ??? // rest on the book
}
```

我们首先假设新进的买入指令价格比提供的最佳出售价格低，这时买入指令就会被加入到指令簿中，即未成交。我们要尝试回答的问题是“这个指令应该被加到指令簿的什么位置？”这个问题的答案是指令簿需要进行一个对数时间的搜索来查找和这个价格相对应的价格水平对象（price level）。从 ListOrderBook 的定义中可以知道 tree 里面的每一个值都表示为指令的一个 List。回想起和交易员主管 Dave 的讨论，在同一个价格水平的指令会按照时间先后进行成交。从概念上讲，价格水平的类型即一个先进先出的队列。由于指令会被增加到队尾，这暗示了增加一个指令到相应价格水平是一个线性时间的操作。以下代码片段可以证实你的猜测：

```
val orders = orderBook.bids.getOrElse(buyOrder.price, Nil)
        orderBook.copy(bids = orderBook.bids + (buyOrder.price ->
orders.:+(buyOrder))) -> LimitOrderAdded(currentTime())
```

这个代码片段显示增加一个未成交指令到指令簿将包含一个线性时间的向后增加元素操作，即增加元素到 BuyLimitOrder 的 List。此时，你会开始疑惑 MV 交易公司如何能在这样的指令簿实现中获利。在跳跃到这个严肃的话题之前，你考虑了指令成交是如何处理的。

假设新进的买入指令价格比提供的最佳出售价格高或者相等，这时这个买入指令就会跨越指令簿被成交。成交时间顺序指示第一个和买入指令对应的出售指令会被成交，翻译为实现即从一个价格水平（price level）中取出第一个出售指令。在产生一笔成交指令时，你意识到用 List 对价格水平建模可以提供常数时间的性能。以下代码片段显示了在执行成交时如何修改价格水平：

```
case (priceLevel, (sell :: Nil)) => (
    orderBook.copy(offers = orderBook.offers - sell.price),
    OrderExecuted(currentTime(), Execution(buy.id, sell.price),
        Execution(sell.id, sell.price))
)
case (_, (sell :: remainingSells)) => (
    orderBook.copy(offers = orderBook.offers +
        (sell.price -> remainingSells)),
    OrderExecuted(currentTime(), Execution(buy.id, sell.price),
        Execution(sell.id, sell.price))
)
```

ListOrderBook 利用 List 的类型匹配的优势来处理两种可能的成交场景：

- 成交时的出售指令是价格水平里面的唯一指令。
- 剩下的出售指令继续在价格水平中存留。

在第一个场景中，offers TreeMap 里的键被移除了，此价格水平也就从指令簿中移除了。在后一个场景中，剩余的指令形成了新的价格水平。显然，指令簿对指令成交而非未成交进行了优化。此时你疑惑为什么在指令簿的实现里面会存在这样的偏见。“也许指令成交比未成交更为普遍？”你不确定是否如此，于是记下来需要和 Dave 讨论此事。



先暂停一会儿，考虑一下你自己设计的系统里面存在的偏见。你有针对系统使用限制或者系统延迟限制去按比例地进行优化

> 吗？回顾自己的系统设计，你的设计选择有将你导向针对最重要的操作达到最好的性能吗？事后回顾让我们可以容易地找出那些（被遗忘的）未做优化考虑的设计选择。反思你如何做的这些决定，你会更有能力避免在将来的系统设计里面引入类似的不足。

取消指令

ListOrderBook 也支持 CancelOrder 命令来通过 ID 移除一个已经存在的指令。取消请求对 ListOrderBook 而言是一个算法上的挑战。由于只提供了指令 ID，ListOrderBook 无法高效地决定此指令在指令簿的买入一边或者卖出一边。为了决定在哪一边，需要清除买入或者出售的价格水平来查找指令 ID。这是一个复杂度与每一边的价格水平数量及每一个价格水平的长度成比例的操作。最差情况是提交了一个指令簿中不存在的指令 ID。一个恶意的交易者甚至可以通过提交一系列不存在的指令 ID 来减慢 MV 交易公司的指令簿操作的速度。你于是记下笔记需要和 Dave 讨论这些恶意交易行为，以及 MV 交易公司可以做什么来防止这些行为。

即使取消操作所引用的指令存在于指令簿中并且其价格水平已经被找出，从指令簿中移除这个待取消的指令依然是一个开销昂贵的操作。取消操作是一个线性时间的操作，它需要遍历指令链表并移除匹配到指令 ID 的那个节点。以下的代码片段展示了在 ListOrderBook 中取消出售指令的实现：

```
orderBook.offers.find { case (price, priceLevel) => priceLevel.exists(_.id
      == idToCancel) }
    .fold[(ListOrderBook, Event)](orderBook ->
      OrderCancelRejected(currentTime(), idToCancel)) {
      case (price, priceLevel) =>
        val updatedPriceLevel = priceLevel.filter(_.id != idToCancel)
        orderBook.copy(offers = updatedPriceLevel.nonEmpty match {
          case true => orderBook.offers + (price -> updatedPriceLevel)
          case false => orderBook.offers - price
        }) -> OrderCanceled(currentTime(), idToCancel)
```

学习这段代码片段之后，对于你而言，为什么取消操作是性能最低的指令簿操作就不足为怪了。为了取消指令，对每一个价格水平会有两次线性时间的遍历操作。第一次遍历

是 exists 操作遍历价格水平列表来决定被取消的指令 ID 是否存在某一价格水平中。一旦找到包含这个指令的价格水平，就会有第二次遍历通过 filter 来更新指令簿的内部状态。

取消操作在 ListOrderBook 中的实现印证了 Scala 的集合操作虽富有表现力但同时也是一把双刃剑。富有表现力的优点表现在取消操作的逻辑简单，容易理解和维护。然而这也使之很容易隐藏运行时的低性能，即从价格水平中移除一个指令需要 2*N 的时间（其中 N 是某一价格水平中的指令数量）。这个简单的示例清楚地显示了，在一个性能敏感的环境里，回头一步，从代码层面考虑使用的数据结构在运行时的开支是非常重要的。

当前的指令簿 - queue 实现

你没有对 ListOrderBook 进行太过严厉的讨伐，因为你从之前的开发经验中知道，一般都是有一些情有可原的情况导致了这样的实现。你将注意力转向当前的指令簿实现，它的代码放在 QueueOrderBook 中。浏览代码之后，你惊奇地发现这个实现除了价格水平数据结构之外，与 ListOrderBook 是一致的：

```
case class QueueOrderBook(
    bids: TreeMap[Price, Queue[BuyLimitOrder]],
    offers: TreeMap[Price, Queue[SellLimitOrder]])
```

这两个实现的唯一区别是使用了 scala.collection.immutable.Queue 取代 List 来表示价格水平。从建模的视角来看，使用一个先进先出队列是有道理的。由于时间顺序指定了指令成交顺序，先进先出队列天然的适合存储未成交指令。你开始怀疑，是否单纯只为建模的目的而用 Queue 取代 List 呢？你心中的问题是“用 Queue 替代 List 到底是如何提升指令簿的性能的呢？”为了理解这个修改，我们需要深入研究一下 Scala 的 Queue 实现。

Queue

这个 Queue 的定义代码片段揭秘了一个有趣而深入的事实：

```
class Queue[+A] protected(protected val in: List[A], protected val out:
    List[A])
```

在尚未深入阅读 Queue 的实现之前，我们发现它使用了两个 List 来管理其状态。在

知道了 ListOrderBook 使用 List 来建模先进先出队列之后，看到使用 List 来建立一个具有不变性的先进先出队列也就不会惊奇了。为了理解入队和出队是如何影响 Queue 的性能的，我们来看一下它们的实现。以下代码片段显示了入队的实现：

```
def enqueue[B >: A](elem: B) = new Queue(elem :: in, out)
```

由于元素是向前插入 in 列表的，入队操作是一个常数时间的操作。回忆一下，在 ListOrderBook 中相似的操作是一个线性运行时性能的增加指令操作。相对于 ListOrderBook，QueueOrderBook 获得了一个清晰的性能上的胜利。下一步我们来考虑出队的实现：

```
def dequeue: (A, Queue[A]) = out match {
    case Nil if !in.isEmpty => val rev = in.reverse ; (rev.head, new
        Queue(Nil, rev.tail))
    case x :: xs => (x, new Queue(in, xs))
    case _ => throw new NoSuchElementException("dequeue on empty queue")
}
```



这个实现显示，出队操作应用于一个空队列时会抛出一个异常。异常对于出队操作是一个不可预料的输出，这会让人觉得这和函数式编程范式有所出入。基于这个原因，Queue 也提供了一个返回 Option 的 dequeueOption 方法。这使得我们需要显示处理空队列的情况，并且使之更容易推理了。我们推荐在所有无法保证在非空队列上面调用 dequeue 的情况下使用 dequeueOption。

由于 in 和 out 的相互作用，deuque 操作比 enqueue 更为复杂。为了理解 Queue 的状态是如何被 dequeue 操作管理的，请看一下以下这个表格。这个表格带我们经历了一系列的入队和出队操作，列举了每一步操作之后 in 和 out 的状态。在看这个表格的同时，请考虑一下在 dequeue 中哪一个模式匹配语句会被执行：

操　作	in	out
enqueue(1)	List(1)	Nil
enqueue(2)	List(1,2)	Nil
enqueue(3)	List(1,2,3)	Nil

续表

操　作	in	out
Dequeue	Nil	List(2,3)
dequeue	Nil	List(3)
enqueue(4)	List(4)	List(3)
dequeue	List(4)	Nil
dequeue	Nil	Nil

由于 enqueue 和 dequeue 调用相互混杂，in 和 out 都保持了状态。在展示的序列的最后，队列返回到它最初的状态（即 in 和 out 都为空）。这个实现的关键在于 Queue 通过延迟从 in 到 out 中传送值来分摊 dequeue 的花费，使之成为常数时间操作。从 in 到 out 进行元素传送通过线性时间的 reverse 操作来保持先进先出的顺序。延迟这个昂贵的操作到 out 为空时才进行，这是惰性求值的一种形式。这也是惰性求值如何用来提升运行时性能的一个示例。

现在你对 Queue 的实现方式有了了解，你应该能推断怎样由 QueueOrderBook 带来性能提升了。下面的表格列举了每种场景下修改一个价格水平的运行时性能：

场　景	ListOrderBook	QueueOrderBook
增加未成交指令	线性时间	常数时间
达成成交	常数时间	分摊常数时间
取消指令	线性时间	线性时间

这个表格显示，理解了 Scala 集合 API 的运行时特征之后，对实现进行小幅修改就可以获得可观的性能提升。回想一下 QueueOrderBook 被引入的时候，我们当时注意到它的接口与 ListOrderBook 相一致，只是内部模块修改为使用类似的 Queue 的操作来替换 List 操作。这是一个相对简单的修改，但可以获取前面展示的这些性能提升。

在看到 QueueOrderBook 在处理指令上的性能优势之后，你非常兴奋，但是还留下一个疑惑，那就是，为了提升取消指令的性能，你还可以做些什么。QueueOrderBook 对于取消指令操作保持了同样的低性能，这对你而言问题依然存在。特别是，最近的市场变化暴露了 MV 交易公司商业利益受损是由指令簿的取消操作的性能缺陷导致的。惰性计算对于处理指令带来了巨大的性能提升，这个原则是否可以用于优化对取消指令请求的处理呢？

通过惰性计算来提升取消操作的性能

队列通过增加状态，延迟并批量进行昂贵操作，提供了高性能的 enqueue 和 dequeue 方法。这个原则也可以用于指令簿。在取消指令的时候，有两个开销昂贵的操作：

- 定位包含待取消指令的价格水平。
- 遍历 Queue 或者 List 来移除被取消的指令。

注意第二个操作，一个启发性的问题是“指令簿如何能通过修改内部状态来延迟这个线性遍历操作？”为了回答这个问题，通常去考查你先前的实现的长处是有帮助的。我们知道在之前任一个指令簿的实现中都存在性能提升之处。考虑这些研究成果，一个可能的优化策略是延迟取消操作直到指令成交发生时才进行。这个方法，即增加一个状态来保存指令取消意图，直到性能允许时才进行移除操作，而不是直接从指令簿的状态中移除指令。这个方法可以像下面这样来实现：

```
case class LazyCancelOrderBook(
   pendingCancelIds: Set[OrderId],
   bids: TreeMap[Price, Queue[BuyLimitOrder]],
   offers: TreeMap[Price, Queue[SellLimitOrder]])
```

LazyCancelOrderBook 增加了一个 scala.collection.immutable.Set 类型的附加状态来管理取消指令的 ID，这些待取消的指令尚未反映到 bids 和 offers 状态中。在深入研究 pendingCancalIds 如何使用之前，让我们来研究一下 Scala 的 Set 实现。

Set

Scala 的 Set 实现既不是一个 ADT，如 List，也不是一个具体实现，如 TreeMap。取而代之，它是一个特质，如下面的定义代码所示：

```
trait Set[A]
```

标准库里面定义 Set 为一个特质是为了支持某些依赖于元素个数的具体实现。Set 的伴生对象定义了对于元素个数为 0 到 4 的 5 个实现。每一个实现都包含了固定个数的元素，如 Set3 的实现如下：

```
class Set3[A] private[collection] (elem1: A, elem2: A, elem3: A)
```

当元素个数很小时，这种手工实现的 Set 运行时性能更好。使用这种技术，增加和移

除操作就返回下一个（元素个数加一的实现）或上一个（元素个数减一的实现）手工实现的实例。例如，考虑对 Set3 的+和-操作：

```
def + (elem: A): Set[A] =
  if (contains(elem)) this
  else new Set4(elem1, elem2, elem3, elem)
def - (elem: A): Set[A] =
  if (elem == elem1) new Set2(elem2, elem3)
  else if (elem == elem2) new Set2(elem1, elem3)
  else if (elem == elem3) new Set2(elem1, elem2)
  else this
```

在 Set4 之后，标准库使用一个名为 HashSet 的实现。这可以从 Set4 的+操作实现中看出：

```
def + (elem: A): Set[A] =
  if (contains(elem)) this
  else new HashSet[A] + (elem1, elem2, elem3, elem4, elem)
```

HashSet 与 TreeMap 类似，都有一个用于管理内部状态的高效的数据结构支持。对于 HashSet，这个支持数据结构是一个散列字典树（hash trie）。根据 Scala 集合性能概述文档（http://docs.scala-lang.org/overviews/collections/performance-characteristics.html）中的描述，散列字典树对增加、移除、包含操作提供分摊常数时间（amortized constant time）的复杂度。如果你想更深入地了解散列字典树是如何工作的，Scala 散列字典树概述（http://docs.scala-lang.org/overviews/collections/concrete-immutable-collection-classes.html#hash-tries）是一个很好的起点。

回到 LazyCancelOrderBook 的实现，我们现在认识到 pendingCancelIds 的常规 set 操作可以在分摊常数时间内完成。倘若我们只关心增加、删除、包含操作，它们会在 set 元素个数增长时存在一个极小的间接开销。我们可以使用 pendingCancelIds 来表示从指令簿中移除指令的意图，而非付出真实的移除操作的代价。这个实现简化了取消指令的处理，使之成为常数时间的增加元素到 pendingCancelIds 的操作：

```
def handleCancelOrder(
  currentTime: () => EventInstant,
  ob: LazyCancelOrderBook,
```

```
    id: OrderId): (LazyCancelOrderBook, Event) =
    ob.copy(pendingCancelIds = ob.pendingCancelIds + id) ->
      OrderCanceled(currentTime(), id)
```

事实上 handleCancelOrder 的实现变得不重要了，因为从指令簿中移除指令的工作被延迟了。虽然这个实现是一个性能上的胜利，但同时它也存在一个严重的缺陷。使用这个实现我们再也不能从指令簿中定位那些不存在的指令 ID 了，而之前这些不存在的指令 ID 会被需要产生 OrderCancelRejected。一种满足这个需求的办法是维护另一个包含指令簿中活跃指令 ID 的 Set。现在 LazyCancelOrderBook 的状态看起来就像下面这样：

```
case class LazyCancelOrderBook(
    activeIds: Set[OrderId],
    pendingCancelIds: Set[OrderId],
    bids: TreeMap[Price, Queue[BuyLimitOrder]],
    offers: TreeMap[Price, Queue[SellLimitOrder]])
```

有了这个定义，我们就能重写 handleCancelOrder 来实现对不存在的指令 ID 的查找需求：

```
def handleCancelOrder(
    currentTime: () => EventInstant,
    ob: LazyCancelOrderBook,
    id: OrderId): (LazyCancelOrderBook, Event) =
    ob.activeIds.contains(id) match {
      case true => ob.copy(activeIds = ob.activeIds - id,
        pendingCancelIds = ob.pendingCancelIds + id) ->
        OrderCanceled(currentTime(), id)
      case false => ob -> OrderCancelRejected(currentTime(), id)
    }
```

当指令 ID 存在时，这个实现包含了 3 个分摊常数时间的操作。一，是存在一个用于判断指令 ID 是否存在于指令簿的操作。然后是指令 ID 从活跃指令 ID 的 Set 中移除，并增加到待取消指令的 Set 中。先前的实现中，这个场景需要两个线性时间的操作。现在，对指令 ID 不存在这种退化场景的处理，就优化为一个分摊常数时间的操作。

先别急着庆祝性能胜利，记住，我们还必须从指令簿里面移除取消的指令。为了降低取消指令的开销，两个潜在的巨大 set 被加入了指令簿中，它们增加了内存消耗及垃圾搜

集压力。另外，我们还需要基准测试来证明理论上的性能提升确实是真实的性能提升。

为了完成 LazyCancelOrderBook 的实现，我们还需要考虑在处理指令时操作 activeIds，以及在产生交易时操作 pendingCancelIds。你可能还记得，处理指令包括两个场景：

- 增加一个未成交指令
- 跨越指令簿产生一笔成交

下面是一段部分实现的代码，它可以为我们准备好处理 BuyLimitOrder 的这两种场景：

```
orderBook.bestOffer.exists(_.price.value <= buy.price.value) match {
      case true => ??? // 成交
      case false => ???  // 未成交
```

为了处理买入指令未成交的情况，新进的买入指令需要加入队列，并且指令 ID 需要加入到 activeOrderIds 集合：

```
  def restLimitOrder: (LazyCancelOrderBook, Event) = {
    val orders = orderBook.bids.getOrElse(buy.price, Queue.empty)
    orderBook.copy(bids = orderBook.bids + (buy.price ->
              orders.enqueue(buy)),
          activeIds = orderBook.activeIds + buy.id) ->
      LimitOrderAdded(currentTime())
  }
  orderBook.bestOffer.exists(_.price.value <= buy.price.value) match {
    case true => ??? // 成交
    case false => restLimitOrder
```

在上面的代码中，增加一个未成交指令的逻辑被抽取为一个名为 restLimitOrder 的方法。这个逻辑与 ListOrderBook 的类似，只是增加了一个分摊常数时间的增加活跃指令 ID 的操作。这个修改直接而简单，并增加了一点点的额外处理时间。最后，我们来考虑更为复杂的指令成交的场景。这个场景跟 Queue.dequeue 类似，需要在它的实现中付出代价处理被延迟的操作。我们面临的第一个难题是：决定哪一个指令可以被成交，哪一个指令由于取消而需要被移除。假设 orderBook 在访问范围内，findActiveOrder 可以提供这个功能，代码如下：

```
  @tailrec
```

```
def findActiveOrder(
    q: Queue[SellLimitOrder],
    idsToRemove: Set[OrderId]): (Option[SellLimitOrder],
      Option[Queue[SellLimitOrder]], Set[OrderId]) =
    q.dequeueOption match {
      case Some((o, qq)) => orderBook.pendingCancelIds.contains(o.id)
        match {
          case true =>
            findActiveOrder(qq, idsToRemove + o.id)
          case false =>
            (Some(o), if (qq.nonEmpty) Some(qq) else None,
              idsToRemove + o.id)
        }
      case None => (None, None, idsToRemove)
    }
```

findActiveOrder递归的检查卖出指令的价格水平，直到找到一个可以成交的指令，或者价格水平为空。在查找一个可以成交的卖出指令之外，这个方法还返回剩余的价格水平。那些被取消的指令 ID，必须从 pendingCancelIds 中移除。在这里，我们看到，之前未处理的取消指令请求，被延迟到了这里进行批量处理。当期间没有取消指令发生时，系统不断处理成交指令，指令成交的处理过程就只是一个分摊常数时间的操作。最差情况是指令成交的处理过程变成一个线性运行时间，这个时间与价格水平中的取消指令数量成比例。让我们来看看 findActiveOrder 是如何用于更新指令簿的状态的：

```
orderBook.offers.headOption.fold(restLimitOrder) {
  case (price, offers) => findActiveOrder(offers, Set.empty) match {
    case (Some(o), Some(qq), rms) => (orderBook.copy(
        offers = orderBook.offers + (o.price -> qq),
        activeIds = orderBook.activeIds -- rms),
      OrderExecuted(currentTime(),
        Execution(buy.id, o.price), Execution(o.id, o.price)))
    case (Some(o), None, rms) => (orderBook.copy(
        offers = orderBook.offers - o.price,
        activeIds = orderBook.activeIds -- rms),
      OrderExecuted(currentTime(),
        Execution(buy.id, o.price), Execution(o.id, o.price)))
    case (None, _, rms) =>
```

```
            val bs = orderBook.bids.getOrElse(buy.price,
                Queue.empty).enqueue(buy)
            (orderBook.copy(bids = orderBook.bids + (buy.price -> bs),
                offers = orderBook.offers - price,
                activeIds = orderBook.activeIds -- rms + buy.id),
            LimitOrderAdded(currentTime()))
        }
    }
```

由于需要移除取消指令，并且需要从pendingCancelIds中移除已经移除的指令ID，成交指令的实现现在无可争议地变得比ListOrderBook或者QueueOrderBook中的更为复杂了。在以上3个模式匹配语句中，指令ID集合作为元组成员返回，从pendingCancelIds中移除它们就表明指令从指令簿中移除了。前两个模式匹配语句的在处理上区别，是一个处理找到了可成交指令并且有一到多个指令在价格水平中存留，另一个处理找到了可成交指令并且剩余价格水平为空。在后一个场景中，该价格水平从指令簿中移除。第3个模式匹配语句负责处理没有找到可成交指令的场景。如果一个可成交指令都没有找到，那是因为所有指令都是被取消指令，这时整个价格水平都被搜索过了，所以此时价格水平为空。

LazyCancelOrderBook 基准测试

作为一个严谨的性能工程师，你知道，虽然代码编译通过了，测试通过了，但工作还尚未完成。你开始深思要如何进行基准测试，才能确定你的修改确实提升了真实运行中的性能。你第一个想法是单独测试取消指令操作来确认这个操作确实被优化了。为了达此目的，你重新查看CancelBenchmarks，它是在第2章“在JVM中度量性能”中引入用来测试QueueOrderBook和LazyCancelOrderBook的。此基准测试测试了在各种大小的价格水平下，取消第一个指令，取消最后一个指令，以及取消一个不存在的指令的性能。由于源代码和之前的一致，在此我们省略代码，重点关注结果。运行下面的命令可以得到这些结果：

```
sbt 'project chapter4' 'jmh:run CancelBenchmarks -foe true'
```

此基准测试给我们提供的结果如下：

基准测试	队列中的指令数	吞吐量（ops）	错误率（占吞吐量百分比）
eagerCancelFirstOrderInLine	1	6,912,696.09	± 0.44
lazyCancelFirstOrderInLine	1	25,676,031.5	± 0.22
eagerCancelFirstOrderInLine	10	2,332,046.09	± 0.96
lazyCancelFirstOrderInLine	10	12,656,750.43	± 0.31
eagerCancelFirstOrderInLine	1	5,641,784.63	± 0.49
lazyCancelFirstOrderInLine	1	25,619,665.34	± 0.48
eagerCancelFirstOrderInLine	10	1,788,885.62	± 0.39
lazyCancelFirstOrderInLine	10	13,269,215.32	± 0.30
eagerCancelFirstOrderInLine	1	9,351,630.96	± 0.19
lazyCancelFirstOrderInLine	1	31,742,147.67	± 0.65
eagerCancelFirstOrderInLine	10	6,897,164.11	± 0.25
lazyCancelFirstOrderInLine	10	24,102,925.78	± 0.24

这个测试证明了在数量从 1 到 10 的价格水平下，取消第一个指令，取消最后一个指令，取消不存在的指令，相比 QueueOrderBook、LazyCancelOrderBook 都更具有性能优势。这和期望结果一致，因为 LazyCancelOrderBook 将最昂贵的操作延迟到了指令成交时执行。我们看到取消操作的性能稳定，没有依赖被取消指令的位置，这还进一步证明了移除指令的工作被延迟了。和期望结果一致的还有，取消不存在指令性能也提升了，因为不再需要一个线性的遍历操作来确定指令是否存在。然而，我们注意到指令数量从 1 增长到 10 时性能受到了影响。我们可以假定 50%的吞吐量下降是由于额外的管理活跃指令 ID 和待取消指令 ID 的开销导致的。

这个结果充满希望地预示了你的修改确实提升了真实运行性能。新的实现顺利通过了石蕊试纸试验，你接下来需要考虑如何有代表性地模拟指令成交和指令取消组合的场景。你决定集中精力创建一个小的基准测试，组合成交和取消来试验 LazyCancelOrderBook，这是更接近生产环境的场景。你回想起最近一次午饭时和 Dave 关于市场交易流程的沟通，他说，一般而言，每一个指令成交的同时会有两个指令取消。基于这个想法，你创建了一个指令成交和取消相互交错场景下的基准测试。对于这两个指令簿的实现，你想要测试如下场景下的性能表现：

- 每两个指令成交有一个指令取消。
- 每一个指令成交有一个指令取消。
- 每一个指令成交有两个指令取消。

这 3 个场景和生产环境类似，它们将帮助揭露 LazyCancelOrderBook 的缺陷。此测试要求将指令簿初始化为包含一些将要取消或者成交的未成交指令。以下代码示例了如何在一个 JMH 测试中初始化指令簿：

```
@State(Scope.Benchmark)
class InterleavedOrderState {
    var lazyBook: LazyCancelOrderBook = LazyCancelOrderBook.empty
    var eagerBook: QueueOrderBook = QueueOrderBook.empty
    @Setup
    def setup(): Unit = {
      lazyBook = (1 to maxOrderCount).foldLeft(LazyCancelOrderBook.empty)
{
          case (b, i) => LazyCancelOrderBook.handle(
              () => EventInstant.now(), b, AddLimitOrder(
                  CommandInstant.now(), BuyLimitOrder(OrderId(i),
bidPrice))).1
      }
      eagerBook = (1 to maxOrderCount).foldLeft(QueueOrderBook.empty) {
          case (b, i) => QueueOrderBook.handle(
              () => EventInstant.now(), b, AddLimitOrder(
                  CommandInstant.now(), BuyLimitOrder(OrderId(i),
bidPrice))).1
      }
    }
}
```

在每个试验之前，两个指令簿都填满了 maxOrderCount（定义为 30）个未成交指令。由于有 3 个测试场景和 2 个指令簿，这次测试一共定义有 6 个基准测试。每个实现的指令簿有相同的 3 组测试场景。为避免重复，以下代码展示了 LazyCancelOrderBook 的 3 个基准测试：

```
@Benchmark
def lazyOneToOneCT(state: InterleavedOrderState): LazyCancelOrderBook = {
    val b1 = LazyCancelOrderBook.handle(() => EventInstant.now(),
      state.lazyBook, firstCancel)._1
    LazyCancelOrderBook.handle(() => EventInstant.now(),
      b1, firstCrossSell)._1
```

```
    }
    @Benchmark
    def lazyTwoToOneCT(state: InterleavedOrderState): LazyCancelOrderBook = {
        val b1 = LazyCancelOrderBook.handle(() => EventInstant.now(),
            state.lazyBook, firstCancel)._1
        val b2 = LazyCancelOrderBook.handle(() => EventInstant.now(),
            b1, secondCancel)._1
        LazyCancelOrderBook.handle(() => EventInstant.now(),
            b2, firstCrossSell)._1
    }
    @Benchmark
    def lazyOneToTwoCT(state: InterleavedOrderState): LazyCancelOrderBook = {
        val b1 = LazyCancelOrderBook.handle(() => EventInstant.now(),
            state.lazyBook, firstCancel)._1
        val b2 = LazyCancelOrderBook.handle(() => EventInstant.now(),
            b1, firstCrossSell)._1
        LazyCancelOrderBook.handle(() => EventInstant.now(),
            b2, secondCrossSell)._1
    }
```

这里的基准测试遵守相同的规则，即先进行取消指令操作（"C"），再进行成交指令操作（"T"）。比如，最后一个测试实现的场景是每两笔成交对应一笔取消。这些命令定义的数据不在程序数据范围内，这可以避免在基准测试过程中生成垃圾数据。运行基准测试的命令如下：

```
sbt 'project chapter4' 'jmh:run InterleavedOrderBenchmarks -foe true'
```

运行产生的结果如下：

基准测试	吞吐量（ops）	错误率百分比
eagerOneToTwoCT	797,339.08	± 2.63
lazyOneToTwoCT	1,123,157.94	± 1.26
eagerOneToOneCT	854,635.26	± 2.48
lazyOneToOneCT	1,469,338.46	± 1.85
eagerTwoToOneCT	497,368.11	± 0.72
lazyTwoToOneCT	1,208,671.60	± 1.69

以上表格显示 LazyCacelOrderBook 的性能优于 QueueOrderBook。延迟处理和

直接处理的性能数据的相对区别展示了一个有趣的关系。从下表中可以一瞥这个相对性能差别：

基准测试	LazyCancelOrderBook 性能提升百分比
OneToTwoCT	141.00%
OneToOneCT	172.00%
TwoToOneCT	243.00%

分析这个表格，我们可以观察到 LazyCancelOrderBook 在一笔成交两次取消的情况下具有最大的性能优势。这个结果证实了延迟处理取消请求所带来的益处。另一个可见的趋势，是随着成交频率增长和取消频率下降，相对 LazyCancelOrderBook 而言，QueueOrderBook 的性能相对提升。这是合理的，因为 LazyCancelOrderBook 在成交操作时引入了额外的开销。在查找取消指令的同时，LazyCancelOrderBook 必须更新 activeIds。QueueOrderBook 避免了这些开销，但是我们看到在性能上，这些额外的开销一直在掩盖 QueueOrderBook 的光彩。总结这些结果，我们对于 LazyCancelOrderBook 取代 QueueOrderBook 有了更多的信心。在有大量取消操作的场景里，LazyCancelOrderBook 取得了显而易见的胜利，在其他场景里也可以和 QueueOrderBook 保持相等。

学到的经验

这一节中，我们使用了 Scala 集合，并明智地采用了延迟计算，来提升 MV 交易公司基础设施中关键组件的性能。在介绍了这几种指令簿的实现之后，你学习到如何使用合适的数据结构提升性能，同时，了解到未经优化的选择会使我们偏离性能之路。这些练习也向你展示了 Scala 的数个集合实现，灵活使用它们你就可以在性能优化时获益。

LazyCancelOrderBook 阐释了在一个性能敏感的环境里，延迟计算是多么有价值的。当面临性能的挑战时，请问自己以下问题，以确定是否可能使用延迟计算。下表列举了每一个问题，以及对于指令簿而言如何回答：

问　题	应用于指令簿
如何分解为独立的任务？	取消指令行为分解为：识别发送给请求方的事件，从指令簿中移除取消的指令。
为什么在当前我要执行所有步骤？	最初，移除指令急切的执行，是由于在建模取消指令的过程时，它是最符合逻辑的方式。

续表

问　题	应用于指令簿
我可以改变限制条件，以让我用不同的方式建模这个问题吗？	理想情况下，我们可以移除的限制，是拒绝不存在指令的情况。但不幸的是，我们无法控制此事。
什么操作在系统中性能最优？	指令成交以及指令未成交加入指令簿，是性能最优的操作。我们平衡了快速的指令成交操作，让其执行从指令簿中移除被取消指令的操作。

和其他方法一样，延迟计算也不是万能的。有必要精心设计基准测试和性能测试来验证改动确实带来了收益。可以论证，`LazyCancelOrderBook` 比 `QueueOrderBook` 的实现更加复杂，随之会带来更高的系统维护成本。我们现在要做的不是修改系统让其更加复杂，而是比这更难的任务，推理变化的指令执行开销将带来怎样的运行时性能状态。就我们已经测试过的场景而言，和 `QueueOrderBook` 相比，`LazyCancenOrderBook` 保持了相同的或者更优的性能。然而，我们只测试了这许多可能的场景中的一小部分，并且我们只使用了指令簿中的单一价格水平。在现实世界中，为了给新的实现建立足够的信心，还需要额外的基准测试和性能测试。

历史数据分析

你完成了不错的工作，希望你在这过程中学习到了有价值的技能。现在正是时候来探索一下 MV 交易公司系统活动的新的一面。交易专家和数据科学家小组为设计表现更好的交易策略，一直不断在学习历史市场数据。到现在为止，公司还不能奢侈地为这个小组分配技术资源。结果就是，这个组一直都在使用笨重的、不可靠的，并且低性能的工具来分析市场数据并精心地建立交易策略。在一个高性能的指令簿的支持下，公司最高优先级的事情是集中精力改进这些实现了的策略。你最好的朋友 Dave，明确地请你加入这个组帮助他们实现基础设施现代化。

滞后时序收益率（lagged time series returns）

这个组使用的主要工具是一个设计简单的，通过历史交易数据计算滞后时序收益率的程序。大家至今都对这个工具非常失望。它不仅返回的大部分结果都是无效的，而且速度慢又很脆弱。深入代码之前，Dave 给你简短地演示了里面的业务规则。时序收益率来自于时间序列（价格）中值。中值每分钟进行计算，基于每一次交易的买入和卖出价格计算而来。作为示例，请看下表：

指令时间	买入价格	卖出价格	中　值
01/29/16 07:45	2.3	2.5	2.55
01/29/16 07:46	2.1	2.4	2.25
01/29/16 07:47	2.9	3.4	3.15
01/29/16 07:48	3.2	3.4	3.3
01/29/16 07:49	3.1	3.3	3.2

计算中值的公式是（买入价格+卖出价格）/ 2。例如，指令时间 01/29/16 07:47 的中值是*(2.9+3.4)/2*，即 3.15。



在真实世界中，中值还会由交易量（the volumne of the transaction）来衡量，并且时间序列会使用更细粒度的时间单元，如秒甚至毫秒。为保持简单，我们忽略了容量维度，假定所有交易量为 1。我们只关注按分钟计算数据点，而不是使用秒甚至毫秒这样更细的粒度。

收益率序列由一系列的中值计算而来。收益率序列定义为数分钟内的某个增长值（rollup）。如计算 3 分钟收益率 t_3，公式为：（时间点 3 的中值 - 时间点 0 的中值）/ 时间点 0 的中值。我们还在结果上乘以 100 以便使用百分数表示。如果我们用之前的中值序列来计算 3 分钟的收益率序列，我们可以得到下表：

时　间	中　值	3 分钟收益率
01/29/16 07:45	2.55	N/A
01/29/16 07:46	2.25	N/A
01/29/16 07:47	3.15	N/A
01/29/16 07:48	3.3	22.73
01/29/16 07:49	3.2	29.69

注意前 3 个中值没有对应的 3 分钟收益率，这是由于没有可以使用的更早的中值数据。

现在你已经熟悉了这个领域，可以看看现在的代码了。从模型定义开始：

```
case class TimestampMinutes(value: Int) extends AnyVal {
   def next: TimestampMinutes = TimestampMinutes(value + 1)
}
case class AskPrice(value: Int) extends AnyVal
```

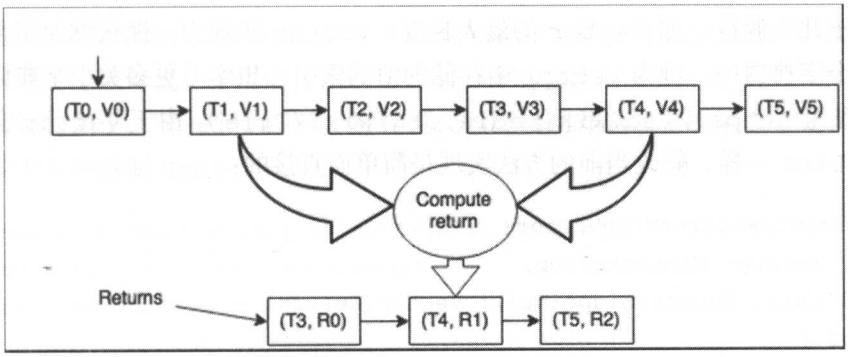
```
case class BidPrice(value: Int) extends AnyVal
case class Execution(time: TimestampMinutes, ask: AskPrice, bid: BidPrice)
case class Midpoint(time: TimestampMinutes, value: Double)
object Midpoint {
   def fromAskAndBid(time: TimestampMinutes,askPrice: AskPrice,
         bidPrice: BidPrice): Midpoint =
      Midpoint(time, (bidPrice.value + askPrice.value) / 2D)
}
case class MinuteRollUp(value: Int) extends AnyVal
case class Return(value: Double) extends AnyVal
object Return {
   def fromMidpoint(start: Midpoint, end: Midpoint): Return =
      Return((end.value - start.value) / start.value * 100)
}
```

一切看起来都很直接。请注意价格、中值和收益率被表示为 Int 和 Double。我们假设我们的系统有能力将价格格式化为整数而不是小数。而且由于我们使用的是元类型 Double 而不是如 BigDecimal 这样的类型，这简化了我们的代码，并且提高了程序的性能。TimestampMinutes 跟更为常用的 UNIX 时间（epoch timestamp）类似，但是时间粒度降低为分钟（参考 https://en.wikipedia.org/wiki/Unix_time）。

学习了模型定义之后，我们来看看 computeReturnsWithList 方法当前的实现：

```
def computeReturnsWithList(
      rollUp: MinuteRollUp,
      data: List[Midpoint]): List[Return] = {
   for { i <- (rollUp.value until data.size).toList }
      yield Return.fromMidpoint(data(i - rollUp.value), data(i))
}
```

这个方法假定输入的中值列表已经按指令时间排序过。这里的实现根据变化的下标随机访问列表，来读取需要的中值数据，以计算出每一个收益率。为计算第二个收益率值（返回列表中下标为 1 的值），用 3 分钟增长值进行计算，我们访问了输入列表中下标为 4 和 1 的元素。下图为如何计算收益率提供了一个可视化的参考：



你曾被提醒过，这个方法不仅慢而且还是不正确的。对于它返回错误结果的事，Dave验证过多次。在解决这个性能问题之前，你必须先解决正确性问题。去优化一个错误的方法，无法利用好你的时间，也即无法利用好公司的钱！很快地，你意识到，这个方法太过相信输入的数据了。为了让这个算法工作，输入的中值列表数据必须满足下列几点：

- 必须被很好地按照指令时间从旧到新排序。
- 每一分钟必须仅有一个中值数据。
- 每一分钟都必须有数据，即没有丢失的数据。

你给 Dave 揭示了这些，以便更深入地了解中值序列是如何生成的。他解释说数据加载于指令簿记录的日志序列。可以确认的是列表是按照时间排序的。并且，他给你保证，考虑到指令簿处理了如此大量级的交易，不可能有某一分钟一个交易都没有。但是，他告诉你更有可能的是同一执行时间会有超过一个中值被计算出来。看上去你找到了引起无效结果的问题所在。解决这个问题应该不会太复杂，现在是时间来看看性能问题了。

我们花了些时间来学习前面一节中的单链列表的结构。你了解到，它为包含列表头尾操作的那些方法进行了优化。与之相反的是，用下标进行随机元素访问，就成为了一个需要线性时间的昂贵操作。为提升中值计算的性能，我们转向另一个对随机访问进行过优化的数据结构：`Vector`。

Vector 向量

为提升系统的性能，我们应该重新考虑存储 `Midpoint` 值的数据结构。一个好的选择是用 `Vector` 替换 `List`，`Vector` 是 Scala 标准库提供的另一个集合类。`Vector` 是一个提供实际常数时间（effectively constant time）随机访问操作的高效集合。随机访问操作的

开销依赖于几个假设，如 Vector 的最大长度。Vector 实现为一棵被称为字典树的排序树。在这个字典树中，键为 Vector 中存储的值的索引（想学习更多关于字典树及其使用场景，请参考 https://en.wikipedia.org/wiki/Trie）。由于 Vector 实现了 Seq 特质，和 List 一样，修改当前的方法实现是简单而直接的：

```
def computeReturnsWithVector(
      rollUp: MinuteRollUp,
      data: Vector[Midpoint]): Vector[Return] = {
   for {
      i <- (rollUp.value until data.size).toVector
   } yield Return.fromMidpoint(data(i - rollUp.value), data(i))
}
```

修改集合类型就足以切换为一个高性能的实现了。为确认确实提升了性能，我们设计了一个简单的基准测试，使用几小时的历史交易数据来测试每一个实现的吞吐量。结果如下：

基准测试	单位时间增长值数量	吞吐量	错误率百分比（相对吞吐量）
computeReturnsWithList	10	534.12	± 1.69
computeReturnsWithVector	10	49,016.77	± 0.98
computeReturnsWithList	60	621.28	± 0.64
computeReturnsWithVector	60	51,666.50	± 1.64
computeReturnsWithList	120	657.44	± 1.07
computeReturnsWithVector	120	43,297.88	± 0.99

Vector 不仅产生了显著的性能优势，还无视增长值的数量，保持了一致的吞吐量。对于索引不变的列表，最好默认使用 Vector 去实现，这是一个通用的准则。实际上 Vector 不仅为随机访问提供了实际常数时间的复杂度，同时为列表首尾操作提供了常数时间的复杂度，如向前及向后增加元素到一个已存在的 Vector。

Vector 的实现是序数（parity）为 32 的树结构。每一个节点被实现为一个大小为 32 的数组，它可以保存 32 个下级子节点的引用或者 32 个值。32 序树结构解释了为什么 Vector 的复杂度是“实际常数”而不是“常数”。这个实现的真实复杂度是 log(32, *N*)，*N* 为 vector 的大小。这被认为是足够接近实际的常数时间。在存储巨大的序列表时，这个集合是一个很好的选择，因为内存被分配为 32 个元素的块。这些块不是对树的每一级预分配的，而是需要时才分配的。

到 Scala 2.10 版本为止，和 List 相比，Vector 的一个缺点是缺少对模式匹配的支持。现在这个问题已经被修复，你可以使用和 List 一样的方法对 Vector 进行模式匹配了。考虑一个简短的例子，对一个 Vector 进行模式匹配访问并返回它的第 3 个元素，或者在元素个数少于 3 时返回 None：

```
def returnThirdElement[A](v: Vector[A]): Option[A] = v match {
    case _ +: _ +: x +: _ => Some(x)
    case _ => None
}
```

在 REPL 中调用这个方法，可以证明模式匹配可以被应用于 Vector，如下所示：

```
scala> returnThirdElement(Vector(1,2,3,4,5))
res1: Option[Int] = Some(3)
```

数据清理

收益率算法现在已经快到令人炫目了。然而，是快到令人炫目地返回不正确的结果！需要记住的是，我们还需要处理一些边界情况，并且对输入数据进行清理。我们的算法仅仅在每分钟有且仅有一个中值时才工作，Dave 却告诉我们是可能见到同一分钟存在多个中值的情况的。

为了处理这个问题，我们创建了一个专用的 MidpointSeries 模块，确保一个包裹一系列无重复 Midpoint 实例的 MidpointSeries 实例，可以被正确地创建出来：

```
class MidpointSeries private(val points: Vector[Midpoint]) extends AnyVal
object MidpointSeries {
    private def removeDuplicates(v: Vector[Midpoint]): Vector[Midpoint] = {
        @tailrec
        def loop(current: Midpoint,
                 rest: Vector[Midpoint],
                 result: Vector[Midpoint]): Vector[Midpoint] = {
            val sameTime = current +: rest.takeWhile(_.time == current.time)
            val average = sameTime.map(_.value).sum / sameTime.size
            val newResult = result :+ Midpoint(current.time, average)
            rest.drop(sameTime.size - 1) match {
              case h +: r => loop(h, r, newResult)
              case _ => newResult
```

```
            }
        }
        v match {
            case h +: rest => loop(h, rest, Vector.empty)
            case _ => Vector.empty
        }
    }
    def fromExecution(executions: Vector[Execution]): MidpointSeries = {
        new MidpointSeries(removeDuplicates(
                executions.map(Midpoint.fromExecution)))
    }
```

我们的 removeDuplicates 方法使用了一个尾递归方法（参考第 3 章内容）。它对所有的中值按时间进行分组，计算这些数据点的均值，创建一个这些均值的新的系列返回。我们的模块提供了一个 fromExecution 工厂方法，从一个 Execution 的 Vector 中创建一个 MidpointSeries 实例。这个工厂方法调用 removeDuplicates 来清理数据。

为改进模块，我们增加了前面的 computeReturns 方法到 MidpointSeries 类中。这样，一旦创建了一个 MidpointSeries 实例，就可以用于计算任意收益率序列了：

```
class MidpointSeries private(val points: Vector[Midpoint]) extends AnyVal
{
    def returns(rollUp: MinuteRollUp): Vector[Return] = {
        for {
            i <- (rollUp.value until points.size).toVector
        } yield Return.fromMidpoint(points(i - rollUp.value), points(i))
    }
}
```

这与我们之前写的是相同的代码，但是这次，我们有信心 points 不会包含重复值了。注意到构造器被标记为 private，所以实例化一个 MidpointSeries 的实例的唯一方法就是通过我们的工厂方法。这保证了不可能创建一个包含脏 Vector 的 MidpointSeries 实例。我们发布了这个程序的一个新版本，希望给 Dave 和他的小组带来好运，然后离开，去吃个很应得的午餐，休息一下。

你回来之后，惊奇地看到 Vanessa，一个数据科学家，等在你的座位前。“收益率序列的代码还是不凑效”，她说。这个小组最终被给到一个可以工作的算法时非常激动，他们决

定不吃午饭来玩玩这个新的程序。不幸的是，他们发现了一些结果的不一致性。你尝试搜集尽可能多的数据，花了一个小时来看这些 Vanessa 提到的无效结果。你注意到它们都包含了有两个特殊符号的交易：FOO 和 BAR。为这些符号记录的交易数据量小，但是很意外，而且通常两次交易之间都会间隔几分钟。关于这些符号你询问了 Dave。Dave 解释说，这些是少量的交易定时检查器的活动（thinly traded tickers），期望的是它们的量级很小。现在问题就清晰了。这些记录的中值序列不满足我们算法的前置条件：每分钟至少有一条交易。你克制自己不去提醒 Dave 他曾经向你保证这种情况是不可能的，而是开始修复这个问题。交易员总是正确的！

你没信心重写（记录日志）算法使其更健壮，同时还要维持系统现在的吞吐量。一个更好的选择是找到一种方法来清洗数据并生成丢失的数据点。你询问了 Vanessa 的建议。她解释道，可以基于附近存在的值来对丢失的数据进行线性插值，这就不会影响交易算法了。你写了一个简短的方法，使用前后的数据点（下面代码中用 a 和 b 表示）对一个特定的时间点计算中值：

```
  private def extrapolate(a: Midpoint,b: Midpoint, time: TimestampMinutes):
Midpoint = {
    val price = a.value +
      ((time.value - a.time.value) / (b.time.value - a.time.value)) *
      (b.value - a.value)
    Midpoint(time, price)
  }
```

有了这个方法，我们就可以编写一个清理数据的方法，放在之前提到的 `removeDuplicates` 函数之后进行数据预处理。

```
  private def addMissingDataPoints(
    v: Vector[Midpoint]): Vector[Midpoint] = {
    @tailrec
    def loop(
      previous: Midpoint,
      rest: Vector[Midpoint],
      result: Vector[Midpoint]): Vector[Midpoint] = rest match {
      case current +: mPoints if previous.time.value == current.time.value
- 1 =>
        // 连续数据点无须插值
```

```
            loop(current, mPoints, result :+ previous)
        case current +: mPoints if previous.time.value < current.time.value
- 1 =>
            // 需要插值
            val newPoint = extrapolate(previous, current, previous.time.next)
            loop(newPoint, rest, result :+ previous)
        case _ => result :+ previous
    }
    v match {
        case h +: rest => loop(h, rest, Vector.empty)
        case _ => Vector.empty
    }
}
```

我们的内部尾递归函数处理了两个数据点连续的情况及有一个数据点丢失的情况。在后一种情况下，我们用`extrapolate`方法创建了一个新的数据点并插入到返回的`Vector`中。注意，我们使用了这个新的数据点对连续丢失的点数据进行插值。我们更新了工厂方法，在移除可能的重复之后执行这个额外的清理工作：

```
def fromExecution(executions: Vector[Execution]): MidpointSeries = {
    new MidpointSeries(
        addMissingDataPoints(
            removeDuplicates(
                executions.map(Midpoint.fromExecution))))
}
```

我们现在可以确认输入数据是干净的，并已准备好用于我们的收益率算法了。

处理多个收益率序列

小组内对于你对系统的改进以及修复现有代码速度印象深刻。他们提到另一个他们已经筹划过一段时间的项目，该项目目前尚不知道该用什么方法实现。几周之前，Vanessa 设计了一个机器学习算法，基于数个定时检查器的收益率序列来计算交易策略。这个算法要求所有的相关收益率序列包含相同数量的数据点。你之前的修改已经处理了这个需求。但是，另一个条件是返回的值必须被规范化过或者说归一化过（scaled）。特征，这个在机器学习里面的术语，表示一个独立可衡量的属性。在我们的例子中，每一个返回的数据点都是一个特征。特征归一化被用于标准化值区间，使得巨大的值区间不会导致学习算法失真。

Vanessa 解释说特征归一化可以帮助她的算法交付更好的结果。我们的程序将需要处理收益率序列的集合，计算一组用于归一化的向量，接着计算新的一组规范化的收益率序列。

数组

在这个系统里面，我们考虑从 Vector 切换到 Array。Array 是一个不变的、有索引的值集合。与 Vector 的实际常数时间实现相比，它为随机访问提供了真正的常数时间复杂度。但是和 Vector 相反的是，Array 是一次性分配的连续块内存。此外，它还不允许进行向前向后增加值的操作。Scala 的数组使用针对内存优化过的 Java 的数组进行实现。Scala 的数组比 Java 本地实现的数组对使用者更友好。大多数在其他 Scala 集合中的可用的方法在使用 Array 时都是可用的。隐式转换也被用于 Array、ArrayOps 及 WrappedArray 中。ArrayOps 是一个对 Array 的简单的包装，增强了 Array，使其具有有序序列的所有操作。在 ArrayOps 上面调用方法会产生一个 Array。与之相反的是，从 Array 到 WrappedArray 的转换是永久性的。在 WrappedArray 上面的调用的转换方法会产生另一个 WrappedArray。我们可以从标准库的文档中看到这些，如下：

```
val arr = Array(1, 2, 3)
val arrReversed = arr.reverse   // arrReversed为Array[Int]
val seqReversed: Seq[Int] = arr.reverse
// seqReversed为WrappedArray
```

决定使用 Array 来实现新模块之后，我们开始编写代码对每一个收益率序列进行特征归一化：

```
class ReturnSeriesFrame(val series: Array[Array[Return]]) {
    val scalingVector: Array[Double] = {
        val v = new Array[Double](series.length)
        for (i <- series.indices) {
            v(i) = series(i).max.value
        }
        v
    }
}
```

对应收益率序列集合的归一化参数列表计算出来了。返回的列表的第一个元素被用来归一化第一个收益率序列，第二个元素被用来归一化第二个收益率序列，以此类推。归一化参数很简单，就是收益率序列里面的最大值。我们现在可以编写代码使用归一化参数列

表来归一化收益率序列帧了：

```
object ReturnSeriesFrame {
    def scaleWithMap(frame: ReturnSeriesFrame): ReturnSeriesFrame = {
        new ReturnSeriesFrame(
            frame.series.zip(frame.scalingVector).map {
                case (series, scaling) => series.map(point =>
Return(point.value / scaling))
            }
        )
    }
}
```

我们将每一个收益率序列和其归一化参数打包，创建了一个新的归一化后的收益率序列返回。我们可以对比一下，此版本用 Array 实现的代码和另一版本用 Vector 实现的代码，两种实现几乎相同（此处代码省略，可以在本书附带的源码中找到）：

基准测试	收益率序列大小	每秒吞吐量	错误率百分比（相对吞吐量）
normalizeWithVector	60	101,116.50	± 0.85
normalizeWithArray	60	176,260.52	± 0.68
normalizeWithVector	1,440	4,077.74	± 0.71
normalizeWithArray	1,440	7,865.85	± 1.39
normalizeWithVector	28,800	282.90	± 1.06
normalizeWithArray	28,800	270.36	± 1.85

这些结果显示，Array 在收益率序列更小时性能比 Vector 更好。随着收益率序列的增大，两者的性能变得旗鼓相当。我们甚至可以看到当收益率序列包含 20 天的数据（28,800 分钟）时，吞吐量变得几乎相等了。对于大型序列而言，Vector 的内存位置及其内存分配模型减小了它和 Array 之间的差距。

我们的实现是符合语言习惯的：使用了高阶函数和具有不变性的数据结构。然而，使用如 zip 和 map 这样的转换函数，会创建新的 Array 的实例。另一个选择是利用 Array 的可变性来限制我们程序产生的（内存）垃圾数量。

使用 Spire cfor 宏编写循环

Scala 支持两种循环结构：for 循环和 while 循环。后者虽然性能好，但是它经常在

函数式编程中被避免使用。它需要使用可变的状态和变量来跟踪循环条件。在这一节中，我们会展示给你一项技术，它利用了 while 循环的性能优势，同时防止了可变的引用泄露到应用的代码中。

Spire 是一个为 Scala 编写的数值型库，它允许开发者编写高效的数值型代码。Spire 利用了一些 Scala 模式，如视图、宏和专门化（回忆第 3 章“释放 Scala 的性能”中关于专门化的部分）。你可以在 https://github.com/non/spire 学习更多关于 Spire 的内容。

cfor 是 Spire 定义的宏之一。它的语法的灵感来自于 Java 中一个传统的 for 循环。在以下的特征归一化实现中，我们使用 cfor 宏来迭代收益率序列并归一化其值：

```
def scaleWithSpire(frame: ReturnSeriesFrame): ReturnSeriesFrame = {
    import spire.syntax.cfor._
    val result = new Array[Array[Return]](frame.series.length)
    cfor(0)(_ < frame.series.length, _ + 1) { i =>
        val s = frame.series(i)
        val scaled = new Array[Return](s.length)
        cfor(0)(_ < s.length, _ + 1) { j =>
            val point = s(j)
            scaled(j) = Return(point.value / frame.scalingVector(i))
        }
        result(i) = scaled
    }
    new ReturnSeriesFrame(result)
}
```

这个例子凸显了 cfor 宏可以被嵌套的特征。这个宏本质上是编译为 Scala 的 while 循环的语法糖。我们可以检查以下生成的字节码来验证这一点：

```
public highperfscala.dataanalysis.ArrayBasedReturnSeriesFrame
scaleWithSpire(highperfscala.dataanalysis.ArrayBasedReturnSeriesFrame);
    Code:
        0: aload_1
        1: invokevirtual #121                  // Method
highperfscala/dataanalysis/ArrayBasedReturnSeriesFrame.series:()[[Lhighp
erf
scala/dataanalysis/Return;
```

```
         4: arraylength
         5: anewarray     #170
   "[Lhighperfscala/dataanalysis/Return;"
         8: astore_2
         9: iconst_0
        10: istore_3
        11: iload_3
     [... omitted for brevity]
        39: iload        6
     [... omitted for brevity]
        82: istore       6
        84: goto         39
     [... omitted for brevity]
        95: istore_3
        96: goto         11
        99: new          #16                 // class
   highperfscala/dataanalysis/ArrayBasedReturnSeriesFrame
       102: dup
       103: aload_2
       104: invokespecial #19                 // Method
   highperfscala/dataanalysis/ArrayBasedReturnSeriesFrame."<init>":([[Lhigh
per
   fscala/dataanalysis/Return;)V
       107: areturn
```

我们注意一下在指令 96 和 84 的两个 goto 语句，它们被用于循环回各自的外循环和内循环开始处（分别从指令 11 和 39 开始）。同样，我们可以针对这个新的实现运行基准测试来确认其性能：

基准测试	收益率序列大小	每秒吞吐量	错误率百分比（相对吞吐量）
normalizeWithArray	60	176,260.52	± 0.68
normalizeWithCfor	60	256,303.49	± 1.33
normalizeWithArray	1,440	7,865.85	± 1.39
normalizeWithCfor	1,440	11,446.47	± 0.89
normalizeWithArray	28,800	270.36	± 1.85
normalizeWithCfor	28,800	463.56	± 1.51

被编译为 `while` 循环的宏有能力提供更好的性能。使用 `cfor` 结构，我们就能在保证性能的同时避免引入多个额外变量。虽然这个方法牺牲了不变性，但可变的范围被限制住了，而且相比使用命令式的 `while` 或 `for` 循环的等价实现而言更加不易出错。

小结

在这一章中，我们探索并尝试了各种各样的集合实现。我们讨论了每一个数据结构的潜在（性能）表现、复杂度以及使用场景。我们还介绍了一个第三方库 Spire，用它来提升程序性能。一些展示过的实现和典型的函数式编程实践虽有所出入，但是我们将变性状态的使用限制在内部模块，对外暴露的接口依然是函数式的。我们猜想你此刻正迫不及待地想学习更多内容，但是在下一章节，我们将会变懒惰起来！跟这一章专注急切集合（eager collections）相反，在下一章中我们将把注意力转向惰性集合。

5

惰性集合及事件溯源

在上一章中，我们探索了一些 Scala 集合，相比惰性计算而言，它们都急切地执行计算。Scala 标准库提供了两类惰性计算的集合：视图和流。为了增加一些探索这些集合的动力，我们会处理另一个在 MV 交易公司遇到的性能难题，它反复出现在为客户生成业绩报表的场景中。在这一章中，我们会覆盖下面这些主题：

- 视图
- 两个真实世界应用中的流式处理
- 事件溯源
- 马尔可夫链的生成

提升用户报表生成速度

你想要学习更多关于 MV 交易公司的客户的知识，于是决定参加业绩汇报周会。环顾一周之后，你发现你是唯一的工程师，其他人都来自于销售团队。Johnny，MV 交易公司客户管理团队的负责人，带你浏览了新签约的客户清单。每一次他念出一个名字的时候，一个很大声的铃声就响起来。对于你，这个动作显得很奇怪，但是销售团队都会激动地欢呼庆祝每一次响铃。

在新客户清单念完，耳边的铃声也停止之后，销售团队的一位同事问 Johnny，“什么时候可以让业绩报表生成得快点啊？交易日的时候，客户每天都给我打电话抱怨不能看到他们的位置和盈亏。我们没有这种透明性，这很尴尬，我们会因此丢失客户的”。你发现这个问题里面的报告是一个 PDF 文件，此文件可以从 MV 交易公司开放给客户的私有 web 接口

下载。除非客户足够有经验可以用 MV 交易公司的业绩 API 来建立他们自己的报表，否则客户只能依赖于这个接口来查阅近期的交易业绩。

你发现这是一个深入理解这个问题的好机会，你问道，“大家好，我来自研发团队。我想在今天坐下来学习更多关于我们客户的知识。你们可以分享更多的关于报表的性能问题吗？我很乐意来帮助定位问题”。在和销售团队交流之后，你了解到这个 PDF 报告是实现一个实时流式网页应用的第一步。这个 PDF 报告使得 MV 交易公司可以迅速地将透析的交易业绩数据传递给客户。每一次客户单击“查看业绩”，就生成一个报表，显示当前客户在过去的一小时、一天和七天中是盈利还是亏损，以此总结客户的业绩趋势。你发现，特别是当市场迅速变化时，客户更有可能会去生成报表。销售团队认为这加剧了这个问题，因为当每个人都尝试查看近期交易业绩时，报表生成变得更慢了。在一些最糟糕的情况下，业绩报表几乎需要十多分钟才能生成，我们的客户却需要获得实时的结果，这对于他们是完全不能接受的。

深入报表生成代码

你迫不及待想深入了解这个问题，于是找到了负责处理报表数据的代码仓库。你浏览了相关领域模型以便理解这些当前出现的问题：

```
case class Ticker(value: String) extends AnyVal
case class Price(value: BigDecimal) extends AnyVal
case class OrderId(value: Long) extends AnyVal
case class CreatedTimestamp(value: Instant) extends AnyVal
case class ClientId(value: Long) extends AnyVal
sealed trait Order {
    def created: CreatedTimestamp
    def id: OrderId
    def ticker: Ticker
    def price: Price
    def clientId: ClientId
}
case class BuyOrder(created: CreatedTimestamp, id: OrderId, ticker: Ticker,
    price: Price, clientId: ClientId) extends Order
case class SellOrder(created: CreatedTimestamp, id: OrderId, ticker:
    Ticker, price: Price,clientId: ClientId) extends Order
case class Execution(created: CreatedTimestamp, id: OrderId, price: Price)
```

在报表上下文中，将指令和成交联系起来对于业绩趋势报表是很重要的，因为这使 MV 交易公司可以识别交易中的盈亏。ClientId（客户 ID）的概念是你之前工作在指令簿或者性能数据分析上时没有接触过的。它被用于识别一个 MV 交易公司的客户账号。由于交易基于客户行为而成交，客户 ID 可以允许我们将一个成交指令和一个客户账号联系起来。

随着逐步检查代码，你识别出了在转换为 PDF 之前，业绩趋势报表的模型表示代码：

```
sealed trait LastHourPnL
case object LastHourPositive extends LastHourPnL
case object LastHourNegative extends LastHourPnL
sealed trait LastDayPnL
case object LastDayPositive extends LastDayPnL
case object LastDayNegative extends LastDayPnL
sealed trait LastSevenDayPnL
case object LastSevenDayPositive extends LastSevenDayPnL
case object LastSevenDayNegative extends LastSevenDayPnL
case class TradingPerformanceTrend(
    ticker: Ticker,
    lastHour: LastHourPnL,
    lastDay: LastDayPnL,
    lastSevenDay: LastSevenDayPnL)
```

对每一个支持的统计时间段，即过去一小时、过去一天、过去七天，**盈亏**（profit and loss, PnL）趋势被表示为不同的抽象数据结构。每一次交易周期内（stock ticker），这 3 个时间段都会被包含到 `TradingPerformanceTrend`。经历多次交易周期后，你推断，客户甚至可以识别出 MV 交易公司是否是在实时计算盈亏。检查了负责计算 `TradingPerformanceTrend` 的 `trend` 方法的签名之后，你可以确认你的猜想：

```
  def trend(
      now: () => Instant,
      findOrders: (Interval, Ticker) => List[Order],
      findExecutions: (Interval, Ticker) => List[Execution],
      request: GenerateTradingPerformanceTrend):
List[TradingPerformanceTrend]
  case class GenerateTradingPerformanceTrend(
      tickers: List[Ticker], clientId: ClientId)
```

计算业绩趋势需要一种方法来决定当前时间，以便在计算每一个时间段的趋势时，可以决定往后查多远。findOrders 和 findExecutions 参数都是函数，分别为在存储的报表数据中查询某一个特定的交易周期内的一段时间内创建的指令和成交的指令。最后一个参数包含了客户的 ID 和报表需要的交易周期数据。每一个时间段趋势由一个泛化了的名为 periodPnL 的内部方法来计算，代码如下：

```
def periodPnL(
    duration: Duration): Map[Ticker, PeriodPnL] = {
    val currentTime = now()
    val interval = new Interval(currentTime.minus(duration), currentTime)
    (for {
        ticker <- request.tickers
        orders = findOrders(interval, ticker)
        executions = findExecutions(interval, ticker)
        idToExecPrice = executions.groupBy(_.id).mapValues(es =>
            Price.average(es.map(_.price)))
        signedExecutionPrices = for {
            o <- orders
            if o.clientId == request.clientId
            price <- idToExecPrice.get(o.id).map(p => o match {
                case _: BuyOrder => Price(p.value * -1)
                case _: SellOrder => p
            }).toList
        } yield price
        trend = signedExecutionPrices.foldLeft(PnL.zero) {
            case (pnl, p) => PnL(pnl.value + p.value)
        } match {
            case p if p.value >= PnL.zero.value => PeriodPositive
            case _ => PeriodNegative
        }
    } yield ticker -> trend).toMap
}
```

periodPnL 方法是一个包含数个逻辑步骤的复杂方法。针对用户提供的每一个交易周期，找出相关的指令和成交数据。为了让指令和成交数据关联起来，使用 groupBy 将一个 OrderId 到 Execution 的映射表创建出来。为了简化之后的计算，事先计算出每一个

成交指令的平均成交价格，以此减少对单个指令到单个值的多次计算。

查找表 idToExecPrice 创建好了之后，下一个逻辑步骤就是过滤掉其他客户的指令。一旦列表中只存留了当前客户的指令，idToExecPrice 就被用于定位成交的指令。最后的两个步骤通过对客户的绝对收入（即盈利和亏损）进行制表来计算出业绩趋势。这些步骤包含了两个增加的领域模型，如下：

```
case class PnL(value: BigDecimal) extends AnyVal
object PnL {
   val zero: PnL = PnL(BigDecimal(0))
}
sealed trait PeriodPnL
case object PeriodPositive extends PeriodPnL
case object PeriodNegative extends PeriodPnL
```

PnL 是一个用于表示客户美金收入的值类（value class）。PeriodPnL 是一个类似于之前引入的可以应用于任意时间段数据的抽象数据结构。这允许 PeriodPnL 可以被复用于过去一小时、一天、七天的业绩趋势计算。

当交易为一个买入时，成交价格变为负数，因为此交易表示现金兑换为股票了。当交易为卖出时，成交价格保持为正，因为交易表示股票兑换为现金。在计算了每一个周期的业绩趋势之后，交易周期列表及 PeriodPnL 元组被转换为一个映射表。

消化了这个实现之后，你开始猜想为什么生成 PDF 成为了一个很耗时的操作。实现中并没有缓存结果的代码，这意味着每次客户发起请求时业绩趋势报表就会被重新计算。随着客户的报表请求的增加，报表生成的等待时间也会相应增加。重新架构报表基础设施来缓存报表对于近期的改动而言太大。反之，你尝试看能不能找到一些可以提升报表生成性能的增量改进。

使用视图提速报表生成

我们在指令簿上面工作时，学习到了 List 类会急切地计算结果。这一性质意味着，在 periodPnL 中，这些脱糖之后为 for 推导（de-sugared for-comprehension）的 filter 和 map 操作，在 orders 上面执行会生成新的列表。即，每一个列表转换都会生成一个新的集合。对于一些拥有大量指令的客户，由于需要迭代 3 次，以及由此引发的对重复生成的列表进行的垃圾搜集，增加了 CPU 时间，也就使之成为一个昂贵的操作。为了改善这一

问题，Scala 提供了一种方法，可以延迟转换元素的时间，直到在下一个流式计算需要时才执行。

从概念上讲，在急切执行的集合上面增加一个视图就可以实现，该视图允许转换使用延迟计算语义来定义。任意 Scala 集合的惰性执行视图都可以通过调用 view 来创建。例如，以下代码片段从一个整数 List 中创建了一个视图：

```
val listView: SeqView[Int, List[Int]] = List(1, 2, 3).view
```

从这个代码片段中，我们可以知道 Scala 使用另一个类型来表示集合的视图，即 SeqView 类型，它有两个类型参数：集合元素类型和集合类型。查看在实例中使用 view 可以使我们更容易理解其在运行时与急切执行的集合的区别。考虑下面代码片段，分别在一个 List 和一个 List 之上的 view 执行相同的操作：

```
println("List evaluation:")
val evens = List(0, 1, 2, 3, 4, 5).map(i => {
    println(s"Adding one to $i")
    i+ 1
}).filter(i => {
    println(s"Filtering $i")
    i % 2 == 0
})
println("--- Printing first two even elements ---")
println(evens.take(2))
println("View evaluation:")
val evensView = List(0, 1, 2, 3, 4, 5).view.map(i => {
    println(s"Adding one to $i")
    i+ 1
}).filter(i => {
    println(s"Filtering $i")
    i % 2 == 0
})
println("--- Printing first two even elements ---")
println(evensView.take(2).toList)
```

此代码片段先执行了一个简单的算法，然后过滤，以找到偶数元素。为了加深我们的理解，代码破坏了函数式编程范式，增加了一个有副作用的 println。列表计算的输出跟

期望一致：

```
List evaluation:
Adding one to 0
Adding one to 1
Adding one to 2
Adding one to 3
Adding one to 4
Adding one to 5
Filtering 1
Filtering 2
Filtering 3
Filtering 4
Filtering 5
Filtering 6
--- Printing first two even elements ---
List(2, 4)
```

使用急切计算方式，在下一个转换之前，每一个转换都一一应用到了每一个元素。现在请考虑以下视图计算的输出：

```
View evaluation:
--- Printing first two even elements ---
Adding one to 0
Filtering 1
Adding one to 1
Filtering 2
Adding one to 2
Filtering 3
Adding one to 3
Filtering 4
List(2, 4)
```

就像我们前面讨论的一样，使用惰性计算时，在需要某一元素之前，没有任何转换会被真正应用。在这个例子中，这意味着值增加和过滤在调用 `toList` 之前都不会发生。“view evaluation”之后没有输出就是没有发生转换的证据。让我们好奇的是，我们还看到了 6 个元素中只有前 4 个被计算了。当向一个视图应用转换的时候，将会对每一个元素应用所有

转换而不是将每一个转换应用到所有元素。通过在一个步骤中应用所有转换，视图能返回前两个（转换后的）元素而不用对整个集合进行计算。由于惰性计算，我们可以从视图的使用看到在这里存在潜在的性能提升。在应用视图的概念到业绩趋势报表之前，让我们先深入看一下视图的实现。

构造一个定制的视图

通过返回一个数据结构组合之前转换的状态和下一个转换，视图就有能力把计算延迟。Scala 的视图实现已被承认是难以理解消化的，因为它提供了大量功能的同时保持了支持所有的 Scala 集合。为了对如何实现视图有一个直观的认识，让我们来构造我们自己的惰性计算视图，该视图将只支持 List 并且只支持 map 操作。作为开始，我们来定义我们的 PseudoView 视图实现所支持的操作：

```
sealed trait PseudoView[A] {
    def map[B](f: A => B): PseudoView[B]
    def toList: List[A]
}
```

PseudoView 定义为一个特质，它支持惰性的从 A 到 B 应用转换，也支持计算所有的转换然后返回一个 List。下一步，我们会定义两种视图类型来支持没有转换被应用时的初始场景，以及支持向一个之前转换过的视图应用另一个转换。函数签名如下代码片段所示：

```
final class InitialView[A](xs: List[A]) extends PseudoView[A]
final class ComposedView[A, B](xs: List[A], fa: A => B) extends PseudoView[B]
```

在这两个场景里，初始的 List 都必须被一直携带着以支持最终转换的应用。在 InitialView 这个基础场景里，由于没有任何转换，所以就没有任何附加的状态。ComposedView 通过携带之前 fa 转换的状态来支持链式计算。

InitialView 的实现直接代理给 ComposedView：

```
final class InitialView[A](xs: List[A]) extends PseudoView[A] {
    def map[B](f: A => B): PseudoView[B] = new ComposedView[A, B](xs, f)
    def toList: List[A] = xs
}
```

以下 List 实现显示了转换是如何通过使用函数组合链接到一起的：

```
final class ComposedView[A, B](xs: List[A], fa: A => B) extends PseudoView[B]
{
    def map[C](f: B => C): PseudoView[C] = new ComposedView(xs,
        f.compose(fa))
    def toList: List[B] = xs.map(fa)
}
```

让我们来构造一个 PseudoView 的伴生对象，以提供 view 的构造，如下：

```
object PseudoView {
    def view[A, B](xs: List[A]): PseudoView[A] = new InitialView(xs)
}
```

我们现在可以编写一个简单的程序来试试 PseudoView 了，此程序证实计算被延迟了：

```
println("PseudoView evaluation:")
val listPseudoView = PseudoView.view(List(0, 1, 2)).map(i => {
    println(s"Adding one to $i")
    i+ 1
}).map(i => {
    println(s"Multiplying $i")
    i* 2
})
println("--- Converting PseudoView to List ---")
println(listPseudoView.toList)
```

运行这个程序，我们可以看到它的输出和使用 Scala 的视图实现相同：

```
PseudoView evaluation:
--- Converting PseudoView to List ---
Adding one to 0
Multiplying 1
Adding one to 1
Multiplying 2
Adding one to 2
Multiplying 3
List(2, 4, 6)
```

PseudoView 帮我们更直观地认识到如何通过建立了一个 Scala 实现视图。以此为起

点，你可以考虑如何支持其他的操作。例如，如何实现 filter？由于 filter 包含了初始的集合，考虑它的实现是很有趣的。从 PseudoView 的定义看，它并没有完善的用于支持 filter 操作的装备，这同时是 Scala 复杂的视图处理的例证。Scala 定义了一个名为 Transformed 的特质来处理这一挑战性问题。Transformed 特质是所有视图操作的基类特质。部分定义如下所示：

```
trait Transformed[+B] extends GenTraversableView[B, Coll] {
    def foreach[U](f: B => U): Unit
    lazy val underlying = self.underlying
}
```

underlying 惰性变量表示初始包装的集合将如何被访问。这与 PseudoView 将 List 状态传入 ComposedView 类似。Transformed 定义了一个具有副作用的 foreach 操作来支持惰性方式的集合操作。使用 foreach 可以让这个特质在实现时修改先前的集合。以下就是 filter 是如何实现的：

```
trait Filtered extends Transformed[A] {
    protected[this] val pred: A => Boolean
    def foreach[U](f: A => U) {
        for (x <- self)
            if (pred(x)) f(x)
    }
}
```

View API 的内部实现使用了 Transformed，用于维护一些必要操作的状态，同时对外的 API 支持和 SeqView 交互。SeqView 遵循另一个在 Scala 集合中常见的模式，通过混入其他的特质继承了大量的操作。SeqView 间接混入了提供访问 Transformed 操作的 TraversableViewLike 特质。

应用视图来提升报表生成性能

有了我们刚刚建立的对视图的直觉，我们可能会从不同的角度看待业绩趋势报表的创建了。Scala 实现的视图使我们可以轻易地从一个急切执行的集合变为惰性执行。如果还能记忆起，指令 ID 到平均成交价格的查找表一旦创建，一系列的转换就会被应用到按周期过滤出的指令上。通过将 orders 转换为视图，就有机会避免不必要的转换，并加快业绩趋势报表的生成速度。

转换为视图是很容易的，确定在哪种情况下，惰性计算表现比急切计算更好则不那么容易了。作为一个优秀的性能工程师，你想要对你提议的修改做一个基准测试。但是你没有权限访问历史指令和成交数据来建立基准测试。于是，你编写了一个微型的基准测试，模拟你正在建模的问题。你正尝试回答的问题是，“对于多大的集合，多少数量的操作下，使用视图来代替 List 才是合理的”？创建视图也是有开销的，因为它需要维护被延迟了的转换的信息，这暗示了它将不会总是性能最优的方案。你写了如下的场景来帮助回答你的问题：

```
@Benchmark
def singleTransformList(state: ViewState): List[Int] =
   state.numbers.map(_ * 2)
@Benchmark
def singleTransformView(state: ViewState): Vector[Int] =
   state.numbers.view.map(_ * 2).toVector
@Benchmark
def twoTransformsList(state: ViewState): List[Int] =
   state.numbers.map(_ * 2).filter(_ % 3 == 0)
@Benchmark
def twoTransformsView(state: ViewState): Vector[Int] =
   state.numbers.view.map(_ * 2).filter(_ % 3 == 0).toVector
@Benchmark
def threeTransformsList(state: ViewState): List[Int] =
   state.numbers.map(_ * 2).map(_ + 7).filter(_ % 3 == 0)
@Benchmark
def threeTransformsView(state: ViewState): Vector[Int] =
   state.numbers.view.map(_ * 2).map(_ + 7).filter(_ % 3 == 0).toVector
```

对于 List 及 Vector 上的视图这两种集合类型，你定义了 3 个测试，来试验转换操作数量逐步增加的场景。使用 Vector 而不是 List 是因为在视图上面的 toList 并不是特定为 List 实现的。就像我们之前见过的一样，List 的操作是为了利用好常数时间复杂度以及性能优异的向前增加元素操作而编写的。toList 执行的是线性时间的向后增加元素操作，它会传递视图是低性能的这样的错误印象。切换为 Vector 就能提供实际常数时间的向后增加元素操作。这个基准测试的状态看上去就像下面这样：

```
@State(Scope.Benchmark)
class ViewState {
```

```
    @Param(Array("10", "1000", "1000000"))
    var collectionSize: Int = 0
    var numbers: List[Int] = Nil
    @Setup
    def setup(): Unit = {
      numbers = (for (i <- 1 to collectionSize) yield i).toList
    }
}
```

ViewState 准备了不同大小的集合来辅助确定视图的性能对于集合大小有多敏感。基准测试通过下面的命令执行：

```
sbt 'project chapter5' 'jmh:run ViewBenchmarks -foe true'
```

执行这个命令会产出以下的结果：

基准测试	集合大小	吞吐量	错误率百分比（相对吞吐量）
singleTransformList	10	15,171,067.61	±2.46
singleTransformView	10	3,175,242.06	±1.37
singleTransformList	1,000	133,818.44	±1.58
singleTransformView	1,000	52,688.80	±1.11
singleTransformList	1,000,000	30.40	±2.72
singleTransformView	1,000,000	86.54	±1.17
twoTransformsList	10	5,008,830.88	±1.12
twoTransformsView	10	4,564,726.04	±1.05
twoTransformsList	1,000	44,252.83	±1.08
twoTransformsView	1,000	80,674.76	±1.12
twoTransformsList	1,000,000	22.85	±3.78
twoTransformsView	1,000,000	77.59	±1.46
threeTransformsList	10	3,360,399.58	±1.11
threeTransformsView	10	3,438,977.91	±1.27
threeTransformsList	1,000	36,226.87	±1.65
threeTransformsView	1,000	58,981.24	±1.80
threeTransformsList	1,000,000	10.33	±3.58
threeTransformsView	1,000,000	49.01	±1.36

关于何种情况使用视图才有更好的性能，这个结果传递给我们一个有趣而深入的认识。

对于很小的集合，例如我们基准测试中的 10 个元素，无论转换操作的数量多少，List 表现更好，而在 1 000 000 个元素时它们间的差距才消失。当转换一个大型的集合时，我们的基准测试中为 1 000 000 个元素，随着转换的数量越多，视图的性能优越性表现越明显。例如，对于 1 000 000 个元素和两个转换，视图的吞吐量几乎是 List 的 3 倍。在中间大小集合的情况下，如例子中的 1 000 个元素，表现则没那么明显。当只有一个转换时，急切执行的 List 表现更好，视图则在超过一个转换时有更好的吞吐量。

随着你的数据量以及转换的数量的增长，视图变得更可能有更好的性能。在这里，你看到了避免中间集合带来的切实的好处。关于性能优化第二个方面可以考虑的是转换的实现。能从更早结束的转换过程获益的转换（如 find）将从惰性计算中获得巨大收益。这一基准测试证实，了解你的数据的大小和意图要执行的转换过程是非常重要的。

视图的注意事项

视图提供了一个简单的方式来优化性能，只需极小的侵入性修改你的系统。易用性是视图诱惑力的一部分，它将吸引你更频繁地使用它们，反之则不然。如前一节中基准测试所示，使用视图并不存在过于琐碎的细节，这意味着默认使用视图可以作为一个次最优的选择。从纯性能的视角来看，还是有其他一些原因需要我们小心谨慎地使用视图的。

SeqView 继承至 Seq

由于视图镜像了集合的 API，这给我们判断何时正在延迟应用转换带来了挑战。基于这个原因，我们建议对视图的使用设置定义良好的边界。工作在客户报表上时，我们将视图的使用限制在了单个内部函数中，并使用急切执行的集合 List 作为返回值类型。最小化系统中使用延迟计算的地方可以在构造运行时执行情况的脑力模型（runtime execution mental model）时减少认知负担。

另一个相关的注解是，需要小心对待视图如何转换为一个急切执行的集合类型，我们觉得这是很重要的。我们代码中的转换是通过调用 toList，它清楚地显示了这个意图。SeqView 还提供了一个 force 方法来强制执行。一个通用的原则是，避免使用 force，因为它返回的是将原始集合类型作为第二个泛型参数进行维护的 scala.collection.immutable.Seq.SeqView，这使得 force 可以在证据充分时（还持有该类型时）返回原始集合类型。然而，某些操作，如 map，会使视图丢失原始集合类型。当这种情况发生时，force 返回一个更通用的 Seq 集合类型。Seq 是一个特质，它是集合库中所有序列类

的父类型，包括视图及另一个我们将要讨论的名为 scala.collection.immutable.Stream 的惰性数据结构。这个继承方案使得以下 3 个语句可以编译：

```
val list: Seq[Int] = List(1, 2, 3)
val view: Seq[Int] = list.view
val stream: Seq[Int] = list.toStream
```

我们相信这不是期望的结果，因为 Seq 数据类型隐藏了关于内部实现的关键信息。它使用一种类型同时表示了惰性和急切执行的集合。考虑以下代码示例，可以理解为什么这不是期望的：

```
def shouldGenerateOrder(xs: Seq[Execution]): Boolean = xs.size >= 3
```

在这个特意加工的示例中，想象一下在一个 Vector 上面调用 shouldGenerateOrder，而且此 Vector 之后会被切换为 SeqView。使用 Vector，确定集合大小是一个常数时间的操作。使用 SeqView，除了可以肯定地说它比使用 Vector.size 更昂贵，你无法通过推理来确定这个操作的运行时情况，所以，由于推理运行时行为很困难并且可能导致无法预料的副作用，应该避免使用 force。

在一个典型的软件系统中，职责区域被分割到离散的模块中。以此业绩趋势报表作为示例，你可以想象一个隔离的模块，它包含转换 List[TradingPerformanceTrend] 为一个 PDF 报表。你也许有兴趣将视图暴露给其他模块以继承惰性转换的好处。如果基准测试证明这种类型的修改是合适的，那么我们建议你选择其中之一。其一，在这种场景下，我们更推荐的选择是使用 Stream，它是 List 的惰性计算版本。我们将会在这一章中迟一些探讨 Stream。其二，如果不能使用 Stream，需要清晰地界定此集合是惰性计算的，以严格限制你的 SeqView 数据类型的使用。

视图不是存储器

在使用视图时还需要考虑的是，清楚地知道转换何时会被接连应用。例如，请考虑如下特意构造的例子，在它关注的场景中，视图被用作多个计算的基础：

```
> val xs = List(1,2,3,4,5).view.map(x => {  println(s"multiply $x"); x * 2
})
xs: scala.collection.SeqView[Int,Seq[_]] = SeqViewM(...)
> val evens = xs.filter(_ % 2 == 0).toList
multiply 1
```

```
multiply 2
multiply 3
multiply 4
multiply 5
evens: List[Int] = List(2, 4, 6, 8, 10)
> val odds = xs.filter(_ % 2 != 0).toList
multiply 1
multiply 2
multiply 3
multiply 4
multiply 5
odds: List[Int] = List()
```

在这个例子中，xs 是一个整数列表的视图。一个将这些整数乘 2 的 map 转换被延迟应用。然后这个视图被用于创建两个 List 实例，一个包含偶数元素，另一个包含奇数元素。我们观察到视图上面的转换被应用了两次，每一次我们将视图转换为列表的时候都会应用。这显示了转换是被延迟应用的，但是计算的结果并没有被缓存。这是需要牢记于心的视图的特征，因为昂贵的转换被多次应用将引起显著的性能下降。这也是为什么应该避免在视图上应用转换的副作用。如果由于某种原因，引用透明没能被支持，由视图的使用带来的副作用和多次执行可能导致软件维护异常困难。

这个例子很直接，视图的错误使用也轻易地暴露出来了。然而，在使用视图时，甚至由标准库提供的方法也可能导致非预期的结果。考虑以下代码片段：

```
> val (evens, odds) = List(1,2,3,4,5).view.map(x => { println(s"multiply
$x"); x * 2 }).partition(_ % 2 == 0)
evens: scala.collection.SeqView[Int,Seq[_]] = SeqViewMF(...)
odds: scala.collection.SeqView[Int,Seq[_]] = SeqViewMF(...)
> println(evens.toList, odds.toList)
multiply 1
multiply 2
multiply 3
multiply 4
multiply 5
multiply 1
multiply 2
multiply 3
```

```
multiply 4
multiply 5
(List(2, 4, 6, 8, 10),List())
```

这个例子的结果与之前的例子一致，但是我们依赖内置的 partition 方法来将原始列表分成两个不同的集合，它们都在原始视图上面执行操作。再一次，我们看到 map 转换在原始的视图上面被应用了两次。这是由 TraversableViewLike 中的 partition 的内部实现所致的。还需记住的是，视图及惰性计算可以帮助提供更好的性能，但是它们应当被小心地对待。一个不错的主意是，在 REPL 中试验并尝试你的算法以确认你正在正确使用视图。

在我们正在运行的业绩趋势报表这个例子中，我们看到了在 Map 上面操作时的一个惰性计算容易忘记的实例。回想此例，存在一个使用以下代码创建的查找表：

```
executions.groupBy(_.id).mapValues(es => Price.average(es.map(_.price)))
```

mapValues 的返回值类型是 Map[A, B]，它并没有在执行策略上面有任何不同的建议。让我们在 REPL 中来运行这个简单的例子：

```
> val m = Map("a" -> 1, "b" -> 2)
m: scala.collection.immutable.Map[String,Int] = Map(a -> 1, b -> 2)
> val m_prime = m.mapValues{ v => println(s"Mapping $v"); v * 2}
Mapping 1
Mapping 2
m_prime: scala.collection.immutable.Map[String,Int] = Map(a -> 2, b -> 4)
> m_prime.get("a")
Mapping 1
res0: Option[Int] = Some(2)
> m_prime.get("a")
Mapping 1
res1: Option[Int] = Some(2)
```

注意到，每次在 m_prime 上面调用 get 来取值的时候，我们都可以观察到转换被应用，甚至在使用同一个键时也是如此。mapValues 会对 map 里面每一个值应用一个惰性执行的转换，类似于一个 map 上面的键的视图操作。这里包含的类型没有提供任何深入的信息，除非你去查看 Map 的实现，或者仔细阅读 mapValues 的相关文档，否则你将可能错过这一重要细节。在使用 mapValues 时，也请考虑使用视图时的注意事项。

打包报表生成结果

在调研 TradingPerformanceTrend 的实现时，我们深入了解了视图及视图如何提高性能。现在我们回到 trend 的实现以完成 List[TradingPerformanceTrend] 的生成。以下代码片段显示 trend 而隐藏了 periodPnL 的实现，因为我们已经完整地评审过 periodPnL 的实现：

```
def trend(
    now: () => Instant,
    findOrders: (Duration, Ticker) => List[Order],
    findExecutions: (Duration, Ticker) => List[Execution],
    request: GenerateTradingPerformanceTrend):
    List[TradingPerformanceTrend] = {
    def periodPnL(start: Instant => Instant): Map[Ticker, PeriodPnL] = { ... }
    val tickerToLastHour = periodPnL(now =>
        now.minus(Period.hours(1).getMillis)).mapValues {
        case PeriodPositive => LastHourPositive
        case PeriodNegative => LastHourNegative
    }
    val tickerToLastDay = periodPnL(now =>
        now.minus(Period.days(1).getMillis)).mapValues {
        case PeriodPositive => LastDayPositive
        case PeriodNegative => LastDayNegative
    }
    val tickerToLastSevenDays = periodPnL(now =>
        now.minus(Period.days(7).getMillis)).mapValues {
        case PeriodPositive => LastSevenDayPositive
        case PeriodNegative => LastSevenDayNegative
    }
    tickerToLastHour.zip(tickerToLastDay).zip(tickerToLastSevenDays)
        .map({
        case (((t, lastHour), (_, lastDay)), (_, lastSevenDays)) =>
            TradingPerformanceTrend(t, lastHour, lastDay, lastSevenDays)
    }).toList
}
```

这个方法主要是将一段时间的 PnL 转换整理为合适时间段的业绩趋势。最后的语句，

包含了两次 zip 调用，将 3 个以 Ticker 为键的映射表和对应阶段的 PnL 业绩趋势值优雅地转化为 List[TradingPerformanceTrend]。zip 在两个集合上面迭代，对集合中的每一个索引创建一个元组。这里有一个简单的代码片段演示了 zip 的用法：

```
println(List(1, 3, 5, 7).zip(List(2, 4, 6)))
```

将产生以下结果：

```
List((1,2), (3,4), (5,6))
```

结果就是对应索引的值被打包到一起了。例如，在索引为 1 处，第一个列表的值是 3，第二个列表的值是 4，产生的元组为(3, 4)。第一个列表有 4 个元素，第二个列表只有 3 个元素，（多出的元素）在结果列表中会被简单地忽略。这一行为有良好的文档描述，但是第一次看到它可能会有点出乎意料。在我们的报表的使用场景中，可以确定每一个键（即每一个 Ticker）都在所有 3 个映射表中出现。在这一场景中，我们可以确定所有 3 个映射表是等长度的。

然而，我们的 zip 用法中有一个不易察觉的 bug。zip 使用了集合的迭代器在元素上面迭代，这暗示了 zip 的使用是对顺序敏感的。3 个映射表都是通过调用 toMap 来创建的，它们都直接代理给 scala.collection.immutable.HashMap 实现的映射表。与 Set 类似，在创建 HashMap 之前，Scala 对于大小较小的映射表提供了一些手写的（handwriting）实现（如 Map2）。到现在，你可能已经意识到这个问题了，HashMap 没有保证顺序。

为了解决这个 bug 并保留 zip 的使用，我们可以利用前面发掘出的 SortedMap，一个由拥有有序键的 TreeMap 支撑的特质。使用 SortedMap 替代 Map 并适当地为 Ticker 定义一个 Ordering，我们就有了一个无 bug 的优雅的交易业绩趋势报表生成解决方案了。由于明智地使用了视图，我们找到了一种方法，用极小的侵入性修改渐进式的改善性能。这将给销售团队一些让他们再次敲响铃声的东西了！同时这也给了我们额外的时间来考虑生成报表的其他方案。

重新思考报表架构

部署了新版本网站之后，在生成业绩报表时就包含了你使用视图进行的优化，你开始想有没有其他可以做的来优化报表生成的性能。突然你灵光一闪，对于某一特定时间段，报表数据是不可变的。针对某一特定小时的 PnL 趋势一旦计算出来就不再更改了。虽然报

表是可变的，但是完全没必要每次用户请求报表都去计算。在想到这一点之后，你想知道每一小时在新的交易数据可用时就生成一个新的报表的困难程度。如果指令和交易事件创建之后作为客户业绩报表生成的输入，它们就可以被即时转换。有了一个已经生成的报表，网站的性能问题就完全消失了，因为报表生成的职责不再属于网站了。

这个新的报表生成策略将我们导向对一个新的设计范式的探索，即事件源。事件源描述的是一种用于系统设计的架构方法，它随着时间来处理事件，而不是依赖于一个现有状态的模型，可以用于回答不同的问题。我们处理的报表系统花了大量的时间来确定完成了交易的指令子集，因为只有当前状态被存储了而事件并没有。想象一下，我们不处理像 `Order` 和 `Execution` 这样的数据，而是随着时间去处理表示系统中发生的事件。一个相关的可能可以处理的事件是 `OrderExecuted` 事件，此事件可以建模如下：

```
case class OrderExecuted(created: CreatedTimestamp, orderId: OrderId, price:
Price)
```

这个事件描述有某件事情发生了，而不是仅仅表示当前状态的快照。扩展此例，想象一下如果 `Order` 还包含了一个可选的 `Price` 来表示成交价格：

```
sealed trait Order {
    def created: CreatedTimestamp
    def id: OrderId
    def ticker: Ticker
    def price: Price
    def clientId: ClientId
    def executionPrice: Option[Price]
}
```

如果这个数据模型被映射到一个关系型数据库，`executionPrice` 将会是一个允许为空的数据库值，在指令成交发生时覆写这个值。当领域模型只反映当前状态时，不变性就丢失了。作为一个函数式的开发者，这个声明应该引起你的注意，因为你懂得不变性提供的推理能力。只保存当前数据状态还可能导致极大的对象，以至于很难编写代码。例如，你该如何表示一个指令被取消了呢？使用当前的做法，最方便的方式是增加一个名为 `isCanceled` 的布尔标记。随着时间流逝，你的系统需求变得更复杂了，你将跟踪更多关于当前状态的特性，`Order` 对象将不断增长。这意味着从数据库加载一个指令对象集到内存中，将会由于增长的内存需求而变得更笨重。如果你有大量的**对象关系映射**（ORM）经

验，你很可能已经经历过这样的困境了。

为了避免膨胀的Order,你也许会尝试分解指令的概念以支持更多的使用场景。例如，如果你仅仅关注成交了的指令，这个模型可能可以将 executionPrice 的数据类型从Option[Price]修改为 Price，并且你也许不再需要布尔值来标记取消状态，因为从定义上讲一个成交了的指令将不可能被取消。

识别出多种定义和表示，对于你之前以为的单一概念而言，是迈向解决我们浏览过的缺陷的重要一步。为扩展这一方法，我们回到事件源的主题。我们可以回放一个事件集来创建 OrderExecuted。让我们轻微地修改一下从指令簿中发出的事件，让其看起来像下面这样：

```
sealed trait OrderBookEvent
case class BuyOrderSubmitted(created: CreatedTimestamp,
   id: OrderId, ticker: Ticker, price: Price, clientId: ClientId)
   extends OrderBookEvent
case class SellOrderSubmitted(created: CreatedTimestamp,
   id: OrderId, ticker: Ticker, price: Price clientId: ClientId)
   extends OrderBookEvent
case class OrderCanceled(created: CreatedTimestamp, id: OrderId)
   extends OrderBookEvent
case class OrderExecuted(created: CreatedTimestamp, id: OrderId,
   price: Price) extends OrderBookEvent
```

如果所有 OrderBookEvents 都持久化了（如到磁盘上），那么就有可能编写一个程序读取所有事件来创建一个 ExecutedOrders 的集合，这可以通过将 BuyOrderSubmitted 和 SellOrderSubmitted 事件与 OrderExecuted 事件关联起来实现。使用这种方法，我们可见的一个优势是，随着时间流逝，我们有可能会问关于系统里面发生过的事情的新的问题，这就可以通过读取事件来简单地回答它们。作为对照，如果一个模型建模的是当前状态，在第一次设计时并没有包含成交信息，就不可能追溯以回答“哪些指令在上一周成交了”这一问题。

我们的新想法令人激动，它有很大潜力会产生巨大的进步。然而，随之而来的也有很多挑战。同上一节中的主要区别是，我们的新使用场景并不是从存储的数据中加载 Order 和 Execution 集合到内存中。而是我们计划在 OrderBookEvent 正被指令簿生成出来时处理它们。从概念上讲，这一方案还是会包含处理一个数据序列。然而，使用之前的方式，

整个数据集在开始任何转换之前就已经存在了。在线处理事件需要设计软件处理尚未被生成出来的数据。我们都清楚的是，无论急切集合还是视图对我们的系统而言都不是好用的工具。幸运的是，Scala 标准库正好给我们提供了一个抽象：Stream。让我们靠近些来观察这个新的集合类型，以更好地理解 Stream 如何可以帮我们实现针对客户业绩报表架构的事件源方案。

Stream 概览

一个流可以被视为一个列表和一个视图的混合。如视图一样，它是惰性计算的，当它的元素被访问或搜集时，转换才会被应用。如列表一样，一个流的元素仅会被计算一次。流常常被描述为一个未发掘的列表，意思是它本质上是一个未被完全评估或者发掘的列表。

在列表可以被双冒号（::）操作符创建的地方，流可以类似地被它自己的操作符创建：

```
> val days = "Monday" :: "Tuesday" :: "Wednesday" :: Nil
days: List[String] = List(Monday, Tuesday, Wednesday)
> val months = "January" #:: "February" #:: "March" #:: Stream.empty
months: scala.collection.immutable.Stream[String] = Stream(January, ?)
```

创建流的语法与创建列表接近。一个不同之处在于返回值。列表立即执行，而流不会。例子中只有第一个元素（"January"）被计算出来了，剩余的值都还是未知的（由一个?符号表示）。

让我们观察一下当我们访问流的一部分时会发生什么：

```
scala> println(months.take(2).toList)
List(January, February)
scala> months
res0: scala.collection.immutable.Stream[String] = Stream(January,
February, ?)
```

通过将其转换为一个列表，我们强制执行了流的前两个元素的计算（请看以下旁注）。前两个月份被打印出来了。然后我们显示 months 的值发现现在第二个元素（"February"）被计算出来了。



在前一个例子中，toList 调用强制 Stream 执行计算。take(2) 是一个惰性转换，还是会返回一个未计算的流：

```
scala> months.take(2)
res0: scala.collection.immutable.Stream[String] = Stream(January, ?)
```

为了突出流的惰性计算特征，我们来看另一个创建流的例子：

```
def powerOf2: Stream[Int] = {
   def next(n: Int): Stream[Int] = {
      println(s"Adding $n")
      n #:: next(2 * n)
   }
   1 #:: next(1)
}
```

这个简短的代码定义了一个函数创建 2 的幂流。它是一个使用首值 1 初始化，尾部由另一个流定义的无穷流。我们增加了一个 println 语句来学习元素的执行时机：

```
scala> val s = powerOf2
s: Stream[Int] = Stream(1, ?)
scala> s.take(8).toList
Adding 1
Adding 2
Adding 4
Adding 8
Adding 16
Adding 32
Adding 64
res0: List[Int] = List(1, 1, 2, 4, 8, 16, 32, 64)
scala> s.take(10).toList
Adding 128
Adding 256
res1: List[Int] = List(1, 1, 2, 4, 8, 16, 32, 64, 128, 256)
```

注意到在我们执行第一个列表转换时，只有前 8 个元素被计算了。在第二次调用时，只有第 9 和第 10 个元素被计算，前 8 个元素作为流的一部分已经被计算了。



以前面的例子为基础，你也许想知道流是否是一个不可变的数据结构。它的完整名为 scala.collection.immutable.Stream，这应该可以给你一个很好的提示。访问流并计算它的一些元素的值导致了对流的修改，这是真实的。然而，这个数据结构依然被认为是不可变的。一是它包含的值一旦赋值就不再修改，二是即或是在被计算之前，这些值在流中也是存在并且有定义的。

前面的例子展示了流的一个有趣属性：有可能创建一个实质上的无限流。使用 powerOf2 创建的流是无界的，始终可以使用我们的 next 方法来创建一个新元素。另一个有用的技术是循环流的创建。一个循环流在它的定义中指向它自己。让我们修改一下之前的例子。我们将允许调用者设置一个初始值，而不是返回 2 的幂的完整序列：

```
def powerOf2(n: Int): Stream[Int] = math.pow(2, n).toInt #:: powerOf2(n+1)
```

math.pow 用于计算 *2^n*。注意我们计算了第一个值，然后使用 powerOf2(n+1) 来定义流的剩余部分，即下一个 2 的幂：

```
scala> powerOf2(3).take(10).toList
res0: List[Int] = List(8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096)
```

流的伴生对象提供了数个工厂方法来初始化一个流。让我们来看看它们中的几个：

- Stream.apply: 这让我们可以创建一个有限值序列流：

```
scala> Stream(1,2,3,4) res0:
scala.collection.immutable.Stream[Int] = Stream(1, ?) scala>
Stream(List(1,2,3,4):_*) res1:
scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

- Stream.fill[A](n: Int)(a: => A): 这会生成一个包含 n 个 a 的流：

```
scala> Stream.fill(4)(10) res0:
scala.collection.immutable.Stream[Int] = Stream(10, ?) scala>
res0.toList res1: List[Int] = List(10, 10, 10, 10)
```

- Stream.from(start: Int): 这会创建一个以 start 开始的递增整数序列：

```
scala> Stream.from(4) res0:
scala.collection.immutable.Stream[Int] = Stream(4, ?) scala>
res0.take(3).toList res1: List[Int] = List(4, 5, 6)
```

我们邀请你看一下伴生对象上可用的其他方法。注意到流也可以直接从列表创建，如下：

```
scala> List(1,2,3,4,5).toStream
res0: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

前面的代码可能会有误导。将一个 List 转换为一个 Stream 并没有付出在内存中计算整个 List 的代价。类似地，如果我们在调用 toStream 之前应用转换（如 map 或者 filter）到 List 上，我们将在整个 List 上面执行这些计算。

跟 List 类似，你可以在流上面应用模式匹配，如下：

```
scala> val s = Stream(1,2,3,4)
s: scala.collection.immutable.Stream[Int] = Stream(1, ?)
scala> s match {
    | case _ #:: _ #:: i #:: _ => i
    | }
res0: Int = 3
```

这个模式匹配从一个流中提取了第 3 个元素。在一个流上进行模式匹配将强制计算那些用于满足匹配表达式需要的元素。上例中，前 3 个元素被计算了。



你可以使用 Stream.Empty 对象，以便在一个空的流上面进行模式匹配。它表示空的流的单例。它的工作方式与 Nil 相对于 List 类似。注意到 Stream 对象包含了一个 empty 方法来返回这个单例；然而，模式匹配需要一个稳定的标识符，所以它就无法在一个有效的 case 语句中调用方法。

事件变换

回到报表系统，我们如何能应用事件源的原理并利用流来改进报表生成方式呢？为了为某一客户计算 TradingPerformanceTrend，我们需要计算 3 个时间段的 PnL 趋势值：

每小时、每天、每 7 天。我们可以编写一个拥有以下签名的方法，它让我们离为每一个趋势值确定 PnL 更近了：

```
def processPnl(e: OrderBookEvent, s: TradeState): (TradeState,
Option[PnlEvent])
```

processPnl 方法接受一个 OrderBookEvent 和一个结构为 TradeState 的状态，产出一个新的 TradeState 及一个可选的 PnlEvent。让我们在查看 TradeState 之前，首先查看 PnlEvent，以便理解这个方法的最终结果：

```
sealed trait PnlEvent
case class PnlIncreased(created: EventInstant, clientId: ClientId,
   ticker: Ticker, profit: Pnl) extends PnlEvent

case class PnlDecreased(created: EventInstant, clientId: ClientId,
    ticker: Ticker, loss: Pnl)extends PnlEvent

case class Pnl(value: BigDecimal) extends AnyVal {
   def isProfit: Boolean = value.signum >= 0
}
object Pnl {
   def fromExecution(buy: Price, sell: Price): Pnl =
      Pnl(sell.value - buy.value)
   val zero: Pnl = Pnl(BigDecimal(0))
}
```

我们看到 PnlEvent 建模了一个表达当客户的 PnL 增加或者减少时的 ADT。使用过去时态来命名这个事件（如增加了的），清楚地表示了这是一个既成事实或某件已经完成的事情的记录。我们还没有看到 TradeState 是如何定义的，也没有看到 processPnl 的实现，但是我们已经可以通过学习这些发出来的事件来推测这一行为了。我们展示了 TradeState 的定义如下，它是将提交了的指令和成交了的指令联系起来所需要的：

```
case class PendingOrder(ticker: Ticker, price: Price,
   clientId: ClientId)
case class TradeState(
   pendingBuys: Map[OrderId, PendingOrder],
   pendingSells: Map[OrderId, PendingOrder]) {
```

```
    def cancelOrder(id: OrderId): TradeState = copy(
        pendingBuys = pendingBuys - id, pendingSells = pendingSells - id)
    def addPendingBuy(o: PendingOrder, id: OrderId): TradeState =
        copy(pendingBuys = pendingBuys + (id -> o))
    def addPendingSell(o: PendingOrder, id: OrderId): TradeState =
        copy(pendingSells = pendingSells + (id -> o))
}
object TradeState {
    val empty: TradeState = TradeState(Map.empty, Map.empty)
}
```

下一步，我们来查看 processPnl 的实现，以观察 PnlEvents 是如何被创建的，如下：

```
def processPnl(
    s: TradeState,
    e: OrderBookEvent): (TradeState, Option[PnlEvent]) = e match {
    case BuyOrderSubmitted(_, id, t, p, cId) =>
        s.addPendingBuy(PendingOrder(t, p, cId), id) -> None
    case SellOrderSubmitted(_, id, t, p, cId) =>
        s.addPendingSell(PendingOrder(t, p, cId), id) -> None
    case OrderCanceled(_, id) => s.cancelOrder(id) -> None
    case OrderExecuted(ts, id, price) =>
        val (p, o) = (s.pendingBuys.get(id), s.pendingSells.get(id)) match {
            case (Some(order), None) =>
                Pnl.fromBidExecution(order.price, price) -> order
            case (None, Some(order)) =>
                Pnl.fromOfferExecution(price, order.price) -> order
            case error => sys.error(
                s"Unsupported retrieval of ID = $id returned: $error")
        }
        s.cancelOrder(id) -> Some(
            if (p.isProfit) PnlIncreased(ts, o.clientId, o.ticker, p)
            else PnlDecreased(ts, o.clientId, o.ticker, p))
}
```

这个实现显示 PnlEvent 是通过模式匹配的来决定事件类型的，程序依照这一规则进行处理。在一个指令提交之后，TradeState 会更新以反映存在一个新的待处理指令，该

指令将被取消或者成交。当指令取消时，这个待处理指令就从 TradeState 中移除。当指令成交发生时，待处理指令被移除，并且额外的处理是，一个 PnlEvent 会在计算交易 PnL 之后发出。交易 PnL 通过比较成交价格和待处理指令的原始价格计算出来。

PnlEvent 提供了足够的信息来为 TradingPerformanceTrend 需要的所有 3 个时间段（小时、天、7 天）计算 PnL 业绩趋势。从 OrderBookEvent 到 PnlEvent 的转换没有副作用产生，使用创建新事件而不是替换当前状态的方式，我们得到了一个不可变的模型。在这些特性的照耀下，processPnl 很容易进行单元测试并且程序的意图很明显。通过明显的展示意图，就有可能和更少的技术上的利益相关者沟通系统如何工作。

遵循类似 State, InputEvent) => (State, Option[OutputEvent] 的方法签名，使用 PnlEvent 作为输入，我们现在可以计算每小时的 PnL 趋势如下：

```
def processHourlyPnl(e: PnlEvent, s: HourlyState): (HourlyState,
    Option[HourlyPnlTrendCalculated])
```

这个签名显示，通过在 HourlyState 中维护状态，就有可能发出 HourlyPnlTrendCalculated 事件。发出的事件定义如下：

```
case class HourlyPnlTrendCalculated(
    start: HourInstant,
    clientId: ClientId,
    ticker: Ticker,
    pnl: LastHourPnL)
```

对某一特定小时、客户 ID 及时钟，HourlyPnlTrendCalculated 是一个表示上一小时 PnL 是正或者负的记录。HourInstant 类是一个值类，它拥有一个伴生对象方法将一个 instant（译注：jotatime 库中的一个 Instant 实例，表示某一不可变的时间）转换为这一小时的开始：

```
case class HourInstant(value: Instant) extends AnyVal {
    def isSameHour(h: HourInstant): Boolean =
        h.value.toDateTime.getHourOfDay == value.toDateTime.getHourOfDay
}
object HourInstant {
    def create(i: EventInstant): HourInstant =
        HourInstant(i.value.toDateTime.withMillisOfSecond(0)
```

```
            .withSecondOfMinute(0).withMinuteOfHour(0).toInstant)
    }
```

让我们看看 HourlyState 是如何定义的，以便更好地理解产生 HourlyPnlTrendCalculated 所需要的状态：

```
case class HourlyState(
    keyToHourlyPnl: Map[(ClientId, Ticker), (HourInstant, Pnl)])
object HourlyState {
    val empty: HourlyState = HourlyState(Map.empty)
}
```

对某一 ClientId 和 Ticker，当前小时的 PnL 存储于 HourlyState。累积着的 PnL 允许 processHourlyPnl 在一小时结束时决定 PnL 趋势。我们现在来检查 processHourlyPnl 的实现来看看 PnlEvent 是如何转换为 HourlyPnlTrend Calculated 的：

```
def processHourlyPnl(
    s: HourlyState,
    e: PnlEvent): (HourlyState, Option[HourlyPnlTrendCalculated]) = {
    def processChange(
        ts: EventInstant,
        clientId: ClientId,
        ticker: Ticker,
        pnl: Pnl): (HourlyState, Option[HourlyPnlTrendCalculated]) = {
        val (start, p) = s.keyToHourlyPnl.get((clientId, ticker)).fold(
            (HourInstant.create(ts), Pnl.zero))(identity)
        start.isSameHour(HourInstant.create(ts)) match {
            case true => (s.copy(keyToHourlyPnl = s.keyToHourlyPnl +
                ((clientId, ticker) ->(start, p + pnl))), None)
            case false => (s.copy(keyToHourlyPnl =
                s.keyToHourlyPnl + ((clientId, ticker) ->
                    (HourInstant.create(ts), Pnl.zero + pnl))),
                Some(HourlyPnlTrendCalculated(start, clientId, ticker,
                    p.isProfit match {
                        case true => LastHourPositive
                        case false => LastHourNegative
                    })))
```

```
            }
        }
        e match {
            case PnlIncreased(ts, clientId, ticker, pnl) => processChange(
                ts, clientId, ticker, pnl)
            case PnlDecreased(ts, clientId, ticker, pnl) => processChange(
                ts, clientId, ticker, pnl)
        }
    }
```

处理增加的和减少的 PnL 遵循同样的流程。名为 processChange 的内部方法处理了完全相同的处理步骤。processChange 通过比较 HourInstant 值决定是否发出 HourlyPnlTrendCalculated。HourInstant 是在最初增加条目时以事件提供的时间戳的小时数为值创建的。当代码中的比较逻辑显示小时数改变了，每小时 PnL 趋势就会由于当前小时已经结束而被计算出来。当小时数没有改变时，提供的 PnL 就被加到状态（即 HourlyState）的 PnL 以继续累积这一小时的 PnL。



这种方案的一个明显的缺点，是当一个客户或者一个时钟没有任何成交的指令时，将没有可能决定小时结束。为了示例简单，我们并没有把时间当作第一位的事件。然而，你可以想象有没有可能建模流逝的时间为一个事件，将其作为 processHourlyPnl 的第二个输入。例如，这一事件可能是下面这样：

```
case class HourElapsed(hour: HourInstant)
```

为了使用这一事件，我们可以修改 processHourlyPnl 的签名来接受一个类型为 Either[HourElapsed, PnlEvent]的事件参数。在一个计时器上面调度 HourElapsed 就能让我们修改 processHourlyPnl 的实现，一旦小时结束就发出 HourlyPnlTrendCalculated 而不是当交易在下一个小时发生才发出。这个例子展示了在你从事件源的角度考虑你的系统时，可以怎样建模时间为领域中明确的部分。

想象一下类似的为每天和每 7 天的 PnL 趋势事件发出事件的方法，然后另一个等待所有 3 个 PnL 趋势事件来产生 TradingPerformanceTrendGenerated 事件的方法，这

些将变得很直接了。最后一个步骤是编写一个有副作用的方法持久化 TradingPerformanceTrend 以使之可以被网站读取。到现在，我们有了一系列转换事件的方法了，但是它们还尚未内聚在一起。下一步，我们将看看如何创建一个管道来转换事件。

请注意，在学习这个示例的过程中，我们实际上并没有计算 PnL。运行一个真实的 PnL 计算将包括更多复杂的算法，还将强制我们引入更多的领域概念。我们选择了一个更为简单的方案，让报告更接近一个曝光的报告（exposure report）。这使得我们可以聚焦在我们想展示的代码和编程实践上。

构建事件源管道

我们使用管道这个术语来表示一个排列好的转换集合，它可能需要多个步骤才能产生出期望的最终结果。这个术语带入我们脑海这样一个图像：一组向多个方向延伸的管子，延伸的路上有弯曲也有转折。我们的目标是编写一个程序，接受 PnlEvents 特质并向标准输出流打印 HourlyPnlTrendCalculated 事件。在一个真实的生成环境中，你可以想象将打印到标准输出流替换为写入一个持久化的数据存储中。每种情况下，我们都在创建一个管道，执行一组引用透明的转换，然后以一个有副作用的操作结束。

管道必须随着新事件的处理累加每次转换的中间状态。在函数式编程范式中，累加常常和一个 foldLeft 操作联合使用。让我们看一个玩具例子（toy example）来更好地理解累加，这个例子对一列整数求和：

```
val sum = Stream(1, 2, 3, 4, 5).foldLeft(0) { case (acc, i) => acc + i }
println(sum) // prints 15
```

这里，我们看到，通过提供一个初始的和值并柯里化一个增加当前元素到累加和值的函数，foldLeft 被应用于计算整数和。acc 经常用作'accumulator'的简称。在这个例子中，累加器和列表元素使用同一种数据类型，即整数。这只不过是一个巧合，它并不是 foldLeft 操作所需要的。这暗示了，累加器类型可以是一个不同于集合元素类型的新类型。

可以使用 foldLeft 作为事件源管道的基础来处理 OrderBookEvent 事件集合，同

时累加中间状态。从前两个处理方法的实现中，我们看到了维护 TradeState 和 HourlyState 的需求。我们定义 PipelineState 来封装需要的状态，如下：

```
case class PipelineState(tradeState: TradeState, hourlyState: HourlyState)
object PipelineState {
   val empty: PipelineState = PipelineState(TradeState.empty,
      HourlyState.empty)
}
```

在 OrderBookEvent 之上调用 foldLeft 时，PipelineState 作为一个累加器，使我们可以为两种转换方法都保存中间状态。现在，我们准备好定义管道的签名了：

```
def pipeline(initial: PipelineState, f: HourlyPnlTrendCalculated => Unit,
   xs: List[OrderBookEvent]): PipelineState
```

pipeline 接受一个初始状态，一个有副作用的函数，该函数在 HourlyPnlTrendCalculated 事件发生时被调用，以及一组 OrderBookEvents 来开始。管道的返回值就是当事件处理完成时管道的状态。让我们看看可以如何利用 foldLeft 来实现管道：

```
def pipeline(
   initial: PipelineState,
   f: HourlyPnlTrendCalculated => Unit,
   xs: Stream[OrderBookEvent]): PipelineState = xs.foldLeft(initial) {
   case (PipelineState(ts, hs), e) =>
      val (tss, pnlEvent) = processPnl(ts, e)
      PipelineState(tss,
         pnlEvent.map(processHourlyPnl(hs, _)).fold(hs) {
            case (hss, Some(hourlyEvent)) =>
               f(hourlyEvent)
               hss
            case (hss, None) => hss
         })
}
```

pipeline 的实现基于在提供的事件上面调用 foldLeft 并以提供的 PipelineState 为累加的起点。提供给 foldLeft 的柯里化函数就是将各个转换连接起来的地方。将两个转换方法和有副作用的事件处理器缝合到一起需要处理几个不同的场景。

让我们来浏览每一种可能的情况以更好地理解管道是怎么工作的。processPnl 方法调用产生一个新的TradeState,并选择性地产生一个 PnlEvent。如果没有 PnlEvent 生成，processHourlyPnl 就不会被调用，前一个 HourlyState 会被返回。如果 PnlEvent 生成出来了，那么 processHourlyPnl 就会执行以决定是否创建 HourlyPnlTrendCalculated。当 HourlyPnlTrendCalculated 生成出来时，接着有副作用的 HourlyPnlTrendCalculated 事件处理器就会被调用并返回新的 HourlyState。如果没有 HourlyPnlTrendCalculated 生成，那么就返回已经存在的 HourlyState。

我们创建了一个简单的例子来证明此管道可以如预期一样的工作，如下：

```
val now = EventInstant(HourInstant.create(EventInstant(
      new Instant())).value)
val Foo = Ticker("FOO")
pipeline(PipelineState.empty, println, Stream(
   BuyOrderSubmitted(now, OrderId(1), Foo, Price(21.07), ClientId(1)),
   OrderExecuted(EventInstant(now.value.plus(
      Duration.standardMinutes(30))), OrderId(1), Price(21.00)),
   BuyOrderSubmitted(EventInstant(now.value.plus(
      Duration.standardMinutes(35))),
      OrderId(2), Foo, Price(24.02), ClientId(1)),

OrderExecuted(EventInstant(now.value.plus(Duration.standardHours(1))),
      OrderId(2), Price(24.02))))
```

在一小时开始时，提交了一个股票 FOO 的买入指令。在这一小时中，买入指令以比买入价更低的价格成交，这指示交易是盈利的。如我们所知，当前的实现依赖后一小时的成交来产生 HourlyPnlTrendCalculated。为了创建这一事件，在第二个小时开始时提交了第二个买入指令。运行这个例子将产生单个 HourlyPnlTrendCalculated 事件输出到标准输出流：

```
HourlyPnlTrendCalculated(HourInstant(2016-02-15T20:00:00.000Z),ClientId(
1),
   Ticker(FOO),LastHourPositive)
```

仅仅使用 Scala 标准库和我们已知的 Scala 集合的知识，我们就成功地创建了一个简单

的事件源管道，尽管这些连接起来的转换多少也被包含了进来。这个例子展示了 foldLeft 强大的功能，可以辅助建立一个事件源管道。使用这个实现，我们就能编写一个功能完整的程序，将预生成的业绩报表写入一个持久化的数据存储中以供网站读取。这个新的设计让我们可以将报表生成的负担从网站的职责中移除，让网站可以提供快速响应的用户体验。这个方案的另一个好处是它将面向领域的语言放置于设计的中心。所有的事件都使用业务术语，并聚焦在建模领域概念上，这使得程序员和利益相关者更容易相互交流。

> 你可能会想了解一个我们尚未提到的和 Stream 有一些相似的特性的数据结构：Iterator。如名字所暗示的，Iterator 提供了在一个数据序列上面迭代的设施。它的简化的定义可以浓缩为如下这样：
>
> ```
> trait Iterator[A] {
> def next: A
> def hasNext: Boolean
> }
> ```
>
> 像 Stream 一样，Iterator 可以避免加载整个数据集到内存中，这使得程序可以编写成仅使用常量的内存。与 Stream 不同的是，Iterator 是可变的并且设计为仅在集合上迭代一次（它扩展了 TraversableOnce 特质）。应该要注意的是，根据标准库文档的描述，在迭代器上面调用了一个方法之后就不能再使用它了。例如，在一个 Iterator 上面调用 size 返回序列的大小，它也消费了整个序列并将 Iterator 的实例设置为无用的状态。和这个准则有区别的仅有 next 和 hasNext 方法。这些属性导致软件难以推理（difficult to reason with），这正好和我们作为函数式程序员所追求的相对立。由于这个原因，我们略过对 Iterator 的深入讨论。

为了进一步探索事件源，我们鼓励你去读一下 Event Store 数据库的文档：http://docs.geteventstore.com/introduction/event-sourcing-basics/。Event Store 是围绕事件源的概念而开发的一个数据库。Event Store 的创建者是 Greg Young，一个在事件源这个主题上很有名的作家。在丰富你关于事件源的理解的同时，请仔细思考在什么时候你相信是适合应用事件源技术的。对于只有简单行为的 CRUD 应用，事件源可能不是值得投入时间的。当你要建模更复杂的行为或者在考虑有严格的性能要求和伸缩性

需求的场景时，在事件源上的投资时间可能会变得合理。例如，就像我们在业绩趋势报表中所看到的一样，从事件源范式的角度去考虑这些性能挑战将向我们暴露一个完全不同的方式去进行设计。

随着你继续探索流式处理的世界，你将发现你希望创建比我们的事件源管道的例子更复杂的转换。为了继续深入流式处理的主题，我们建议你去研究两个相关的库：`akka streams` 和 `functional streams`（前身为 `scalaz-stream`）。这些库提供了相关工具以使用与 `Stream` 不同的抽象去创建更复杂的转换管道。结合对 Event Store 的学习，将加深你对于事件源如何绑定到流式处理里的理解。

马尔可夫流式链

使用前一节最后部分的那个简单程序，我们演示了可以将一系列事件转换连接成一个管道。作为一个善意的（well-intentional）工程师，你希望开发一个可以证明管道正常工作的自动化测试。一种方法是增加一些历史产品数据的样例到初始库中以创建测试。这通常是一个好的选择，但是你担心样例没有大到可以表现大量广泛的场景。另一个选择是编写一个可以创建类似产品数据的事件生成器。这一方案需要更多的前期投入，但是它产出的是一个更有活力的测试管道的方式。

最近的一次和 Dave 关于马尔可夫链的午饭交流让使用生成的数据来测试事件源管道这个想法火花闪耀。Dave 描述了如何让马尔可夫链成为一个只依赖当前状态来决定下一状态的状态转换统计模型。Dave 将股市的状态描绘成一个马尔可夫链，这使他可以基于他觉察到的股市处于上升状态、下降状态或平稳状态来构建交易策略。在阅读了马尔可夫链的维基百科页面之后，你想到基于马尔可夫链来编写一个事件生成器。

我们的最终目标是能生成无限数量的遵循类似产品环境的模式的 `OrderBookEvent`。例如，我们从前面的经验中知道指令取消比指令成交成比例更多，尤其是在易变的市场环境中。这个事件生成器应该要能表现各种事件发生的可能。由于马尔可夫链只依赖它的当前状态去确定它的下一状态，流天然地适合这一场景，因为我们只需要检查当前元素去决定下一个元素。对于我们描述的马尔可夫链，需要确定从当前状态转换到任一其他可能状态的可能性。下表展示了一个可能的情况的集合：

当前状态	买入概率	卖出概率	成交概率	取消概率
BuyOrderSubmitted	10%	15%	40%	40%
SellOrderSubmitted	25%	10%	35%	25%
OrderCanceled	60%	50%	40%	10%
OrderExecuted	30%	30%	55%	30%

这个表格定义了在给定当前状态 OrderBookEvent 时接受到 OrderBookEvent 的可能性。例如，给定一个卖出指令被提交的状态，有 10%的概率在下一步看到第二个卖出指令，35%的概率发生指令成交。我们可以根据我们想要在管道中模拟的市场状态来开发状态转换可能性表。

我们可以使用下述领域模型来建模状态转换：

```
sealed trait Step
case object GenerateBuy extends Step
case object GenerateSell extends Step
case object GenerateCancel extends Step
case object GenerateExecution extends Step
case class Weight(value: Int) extends AnyVal
case class GeneratedWeight(value: Int) extends AnyVal
case class StepTransitionWeights(
    buy: Weight,
    sell: Weight,
    cancel: Weight,
    execution: Weight)
```

在这一领域中，Step 是一个建模可能的状态的抽象数据结构。对于一个给定的 Step，我们将结合 StepTransitionWeights，根据提供的权重来定义转换到不同状态的可能性。GeneratedWeight 是一个值类型，定义了从当前 Step 生成的权重。我们将使用 GeneratedWeight 来驱动从一个 Step 转换到下一个 Step。

下一步，如我们所说，是使用我们的领域根据我们定义的可能性来生成事件。为了使用 Step，定义了一个表示要用的马尔可夫链状态的结构，如下：

```
case class State(
    pendingOrders: Set[OrderId],
    step: Step)
```

这个马尔可夫链需要知道表示为 step 的当前状态。另外，通过在 pendingOrders 中维护提交了的但未被取消或者成交的指令集，我们在马尔可夫链上增加了一个迂回状态。之所以需要这个额外状态，有两个理由。第一，生成取消和成交事件需要连接到某一个已知的指令 ID。第二，限制了我们的马尔可夫链的表示，需要在创建一个取消或成交事件前至少存在一个待处理指令。如果没有待处理指令，转换到一个生成 OrderCanceled 或者 OrderExecuted 的状态是无效的。

使用 State，我们可以写一个有如下签名的方法来管理状态转换：

```
def nextState(
    weight: StepTransitionWeights => GeneratedWeight,
    stepToWeights: Map[Step, StepTransitionWeights],
    s: State): (State, OrderBookEvent)
```

给定一种从当前 StepTransitionWeights 生成权重的方法，一个从 Step 到 StepTransitionWeights 的映射，以及当前的 State，我们就能生成一个新的 State 和一个 OrderBookEvent。为了简单起见，我们略过 nextState 的实现因为我们想要最专注地聚焦在流式处理上面。从这个签名中，我们对于使用这个方法有了足够的理解，但是我们鼓励你去查看代码来填补你在理解上面的缺失。

nextState 方法在我们的马尔可夫链实现中是状态转换的驱动器。我们现在可以基于转换概率表，使用 Stream 的一个便捷的方法 iterate 来生成一个无限的 OrderBookEvent 流。从 Scala 的文档中知道，iterate 通过反复在初始值上面应用一个函数来产生一个无限流。让我们看看如何使用 iterate：

```
val stepToWeights = Map[Step, StepTransitionWeights](
    GenerateBuy -> StepTransitionWeights(
        Weight(10), Weight(25), Weight(40), Weight(40)),
    GenerateSell -> StepTransitionWeights(
        Weight(25), Weight(10), Weight(40), Weight(25)),
    GenerateCancel -> StepTransitionWeights(
        Weight(60), Weight(50), Weight(40), Weight(10)),
    GenerateExecution -> StepTransitionWeights(
        Weight(30), Weight(30), Weight(60), Weight(25)))
val next = State.nextState(
    t => GeneratedWeight(Random.nextInt(t.weightSum.value) + 1),
```

```
    stepToWeights, _: State)
println("State\tEvent")
Stream.iterate(State.initialBuy) { case (s, e) => next(s) }
    .take(5)
    .foreach { case (s, e) => println(s"$s\t$e") }
```

通过以一个 Step 到 StepTransitionWeights 的映射为基础调用 State.nextState 方法，这个代码片段创建了一个马尔可夫链来生成各种各样的 OrderBookEvent。State.nextState 是部分应用的（partially applied），剩下的是未被应用的当前状态。next 函数签名为 State => (State, OrderBookEvent)。有了这个脚手架，通过调用 next，Stream.iterate 被用于生成 OrderBookEvent 的无限序列。与 foldLeft 相似，我们提供了一个初始值 initialBuy 作为迭代的开始，它的定义如下：

```
val initialBuy: (State, OrderBookEvent) = {
    val e = randomBuySubmitted()
    State(Set(e.id), GenerateBuy) -> e
}
```

运行这个代码将产生类似如下的输出：

```
    State = State(Set(OrderId(1612147067584751204)),GenerateBuy)
    Event =
BuyOrderSubmitted(EventInstant(2016-02-22T23:52:40.662Z),OrderId(16121470675
84751204),Ticker(FOO),Price(32),ClientId(28))
    State =
State(Set(OrderId(1612147067584751204),OrderId(7606120383704417020)),Generat
eBuy)
    Event =
BuyOrderSubmitted(EventInstant(2016-02-22T23:52:40.722Z),OrderId(76061203837
04417020),Ticker(XYZ),Price(18),ClientId(54))
    State =
State(Set(OrderId(1612147067584751204),OrderId(7606120383704417020),
OrderId(5522110701609898973)),GenerateBuy)
    Event =
BuyOrderSubmitted(EventInstant(2016-02-22T23:52:40.723Z),OrderId(55221107016
09898973),Ticker(XYZ),Price(62),ClientId(28))
```

```
      State =
State(Set(OrderId(7606120383704417020),OrderId(5522110701609898973)),Generat
eExecution)
      Event =
OrderExecuted(EventInstant(2016-02-22T23:52:40.725Z),OrderId(161214706758475
1204),Price(21))
      State =
State(Set(OrderId(7606120383704417020),OrderId(5522110701609898973),
OrderId(5898687547952369568)),GenerateSell)
      Event =
SellOrderSubmitted(EventInstant(2016-02-22T23:52:40.725Z),OrderId(5898687547
952369568),Ticker(BAR),Price(76),ClientId(45))
```

当然，每次调用会为 GeneratedWeight 生成随机值，即用于下一次转换中选择某个选项的概率，每次调用的结果也就不一样了。这个代码片段提供了为报表基础设施构建大规模测试的基础。从这个例子中，我们看到了一个有趣的马尔可夫链的应用，它被用于在无须访问大量产品数据时，生成表示各种市场情况的事件。我们现在可以编写测试来确认报表基础设施是否能在不同的市场条件下正确地计算 PnL 趋势了。

流的注意事项

流有很多好处，但是也应该被小心使用。在这一节中，我们提到了一些主要的 Stream 的注意事项，以及如何避免它们。

流是存储器

视图不会缓存计算结果，所以每次访问时都重新计算每一元素的值，但是流会存储它的元素的最终形式。一个元素仅会被计算一次，即当第一次访问它时。虽然这是一个避免多次计算同一结果的很好的特点，它也可能导致大量的内存开销，直到某一时刻你的程序最终将内存用尽。

为了避免流存储数据，好的实践是避免保存一个流到一个 val 里。使用 val 将创建一个指向流的头部的永久引用，这会保证每一个计算出来的元素都被缓存。如果一个流被 def 定义，一旦流不再需要时，它就可以作为垃圾搜集掉。

存储可能发生在当调用流上面定义的某些方法时。例如，drop 或者 dropWhile 将执

行并存储直到被丢弃元素的所有的中间元素。这些元素会被存储是由于这些方法是在一个流的实例上面定义的（并且流拥有一个指向它自己的头部的引用）。我们可以实现我们自己的 drop 函数来避免在内存中缓存中间元素：

```
@tailrec
def drop[A](s: Stream[A], count: Int): Stream[A] = count match {
  case 0 => s
  case n if n > 0 => drop(s.tail, count - 1)
  case n if n < 0 => throw new Exception("cannot drop negative count")
}
```

我们通过在 count 值上面进行模式匹配来知道是否可以返回给定的流或者需要在尾部执行递归调用。我们的方法是尾递归的。由于尾递归函数会在每次迭代时会回收它的引用，这确保了我们不会保存到流的头部的引用。s 引用将仅指向流的剩余部分，而不是头部。

另一个有问题的方法的例子是 max。调用 max 将存储流的所有元素以决定哪一个是最大的。让我们来实现一个安全的 max 版本，如下：

```
def max(s: => Stream[Int]): Option[Int] = {
  @tailrec
  def loop(ss: Stream[Int], current: Option[Int]): Option[Int] = ss match
{
    case Stream.Empty => current
    case h #:: rest if current.exists(_ >= h) => loop(rest, current)
    case h #:: rest => loop(rest, Some(h))
  }
  loop(s, None)
}
```

这一次，我们使用一个内部的尾递归函数来暴露一个友好的 API。将当前最大值表示为一个 Option[Int]，以便处理在一个空的流上面调用方法的情况。注意到 max 接受 s 为一个传名参数。这是很重要的，因为反之我们将会在调用内部的尾递归 loop 之前保存一个到流的头部的引用。另一个可能的实现如下：

```
def max(s: => Stream[Int]): Option[Int] = {
  @tailrec
```

```
    def loop(ss: Stream[Int], current: Int): Int = ss match {
        case Stream.Empty => current
        case h #:: hs if h > current => loop(hs, h)
        case h #:: hs if h <= current => loop(hs, current)
    }
    s match {
        case Stream.Empty => None
        case h #:: rest => Some(loop(rest, h))
    }
}
```

这一实现更为简单，这无须争论。我们在 max 函数中检查流是否为空，这让我们要么直接返回 None，要么用一个有效的默认值（流的第一个元素）调用 loop。loop 不再需要处理 Option[Int]了。然而，这个例子并没有实现避免存储的目标。模式匹配将导致 rest 保存一个指向原流的整个尾部的引用，这将会阻止对中间元素的垃圾回收。一个好的实践是仅在一个消费流的尾递归方法中使用模式匹配。

流可以是无限的

从概述中我们看到是有可能定义一个无限流的。然而，在和无限流一起工作时，你需要小心。一些方法可能导致计算整个流，以至于出现 OutOfMemoryError。一些方法很明显，如 toList，它会尝试将整个流存储到一个 List 中，导致对所有元素进行计算。其他一些方法更为微妙。例如，流有一个与 List 上的定义类似的 size 方法。在一个无限流上面调用 size 将导致程序用尽内存。类似地，max 和 sum 将会企图计算整个序列并导致系统崩溃。这一行为尤其危险，因为流是扩展 Seq 的，而 Seq 是序列的基特质。考虑以下代码：

```
def range(s: Seq[Int]): Int = s.max - s.min
```

这个简短的方法使用 Seq[Int]作为唯一参数，返回它的值区间，即最大和最小元素之差。由于流扩展至 Seq，以下调用是有效的：

```
val s: Stream[Int] = ???
range(s)
```

编译器将开心并迅速地生成代码的字节码。然而，s 可能被定义为一个无限流：

```
val s: Stream[Int] = powerOf2(0)
range(s)
java.lang.OutOfMemoryError: GC overhead limit exceeded
    at .powerOf2(<console>:10)
    at $anonfun$powerOf2$1.apply(<console>:10)
    at $anonfun$powerOf2$1.apply(<console>:10)
    at scala.collection.immutable.Stream$Cons.tail(Stream.scala:1233)
    at scala.collection.immutable.Stream$Cons.tail(Stream.scala:1223)
    at scala.collection.immutable.Stream.reduceLeft(Stream.scala:627)
    at
scala.collection.TraversableOnce$class.max(TraversableOnce.scala:229)
    at scala.collection.AbstractTraversable.max(Traversable.scala:104)
    at .range(<console>:10)
    ... 23 elided
```

由于 max 和 min 的实现，range 的调用永远不会返回。这一例子演示了我们在本章前期提到的一个好的实践。

小结

在这一章节中，我们探索了两类 Scala 标准库提供的延迟计算的集合：视图和流。我们探索了它们的特征及实现细节，还包括需要在使用这些抽象时谨记于心的限制。利用你新获得的知识，你定位了一个关键的性能问题，这个问题影响了 MV 交易公司的客户查看他们的业绩趋势。

在流的那一节中，我们趁机将流式处理的概念和事件源联系起来了。我们简单地浏览了事件源范式并介绍了一个简单的事件驱动管道，来优化报表系统的架构，并定义了一个更强大的领域模型。最后，构建了一个马尔可夫链事件生成器来辅助测试我们生成报表的新方案。

通过对急切集合和惰性集合的探索，你现在牢固地掌握了关于 Scala 标准库提供的集合如何工作的知识。在下一章中，我们会继续对 Scala 概念进行探索，从函数编程范式的角度深入了解并发。

6

Scala 的并发

在本章里，我们将把关注的焦点从集合转移到另一个主题：并发。发挥硬件提供的全部 CPU 资源的能力对于开发高性能软件非常重要。不幸的是，写并发代码并非易事，因为很容易写出不安全的程序。如果你有 Java 经验，也许还对 `synchronized` 块和锁心有余悸！`java.util.concurrent` 包归功于很多工具来简化并发代码的开发，但是设计稳定可靠的并发应用仍然是一个严峻的挑战。在本章里，我们会介绍 Scala 标准库提供的并发开发工具。快速介绍完 Scala 并发的主要抽象——`Future`——之后将研究它的特点和我们应当避免的陷阱。本章最后，将探索 `Future` 之外的一个备选方案，叫作 `Task`，它是由 Scalaz 库提供的一个工具。在本章中，我们将探索以下主题：

- 并发 vs 并行
- 使用 Future 的考量
- 块调用和回调
- Scalaz Task

并行回测（backtesting）策略

数据科学家正在用你为他们构建的数据分析工具研究股票交易策略。但是他们现在遇到了障碍，因为回测策略的成本越来越高了。随着他们构造出需要更多历史数据和更多有状态算法的更精妙的交易策略，回测花的时间越来越长。你再一次被召集发挥 Scala 和函数式编程范式的能力来帮助 MV 交易公司提升软件性能。

数据科学家渐进地构建了一个回测工具，使他们能够通过重放（replay）历史数据来判

断交易策略的性能。其机制是提供预先设置的待运行策略，用于测试的时钟（the ticker to test against）和历史数据的时间间隔来重放。回测工具加载市场数据然后应用交易策略来生成交易决策。一旦回测结束，工具会给出总结并显示交易策略的表现结果。在把构思的交易策略部署到生产环境上之前，需要严格地用回测工具来测试。

为了开始了解回测工具，开始阅读代码，如下：

```
sealed trait Strategy
case class PnL(value: BigDecimal) extends AnyVal
case class BacktestPerformanceSummary(pnl: PnL)
case class Ticker(value: String) extends AnyVal

def backtest(
  strategy: Strategy,
  ticker: Ticker,
  testInterval: Interval): BacktestPerformanceSummary = ???
```

阅读了上面的代码后，你找到了驱动回测的主要方法。给定 Strategy、Ticker 和 Interval，它能够生成 BacktestPerformanceSummary。扫描代码库时，你发现了一个文件叫作 CrazyIdeas.scala，这个文件只有 Dave 一个提交者。你在这里看到了调用回测工具的例子：

```
def lastMonths(months: Int): Interval =
   new Interval(new DateTime().minusMonths(months), new DateTime())
backtest(Dave1, Ticker("AAPL"), lastMonths(3))
backtest(Dave1, Ticker("GOOG"), lastMonths(3))
backtest(Dave2, Ticker("AAPL"), lastMonths(3))
backtest(Dave2, Ticker("GOOG"), lastMonths(3))
```

回测工具的用法给了你一个提高性能的可能的线索。看上去当 Dave 有了新想法时，他想对多个股票代码做测试，然后和别的交易策略的表现做比较。目前回测是按顺序执行的。提供回测性能的一种方法是并行执行所有的回测。如果回测的每个调用能够并行而且硬件资源又足够，那么同时回测多个交易策略和股票代码应该会快一些。为了掌握如何并行回测，我们首先需要深入异步编程这个主题来看看 Scala 提供了哪些工具来支持并发编程。



在深入代码前，我们需要丰富用来讨论异步编程的词汇表。并发（Concurrency）和并行（Parallelism）经常被当作同义词使用，但是这两个术语之间存在重要差别。并发指两个（或多个）任务在重叠的时间里启动和执行。同一时间多个任务都在进展中（也就是说，running），但是在一个时间瞬间只有一个任务在实际做事情。当你在单核机器上写并发代码时就是这样的。同一时间只有一个任务能推进，但是多个任务并发地在运行。只有多个任务真正地同时运行时才叫作并行。对于一台双核的机器，你可以同时执行两个任务。基于这个定义，我们可以看到并行依赖于可用的机器。这意味着，可以把程序写成并发的，但是否能并行超出了软件的控制范围。为了更好地理解这些概念，考虑粉刷房间的例子，如果只有一个粉刷匠，粉刷匠可以在第一面墙上粉刷一条，然后在第二面墙上再粉刷一条，然后回到第一面墙，然后再完成第二面墙。粉刷匠确实在同时粉刷两面墙，但同一时刻只能粉刷一面墙。如果有两个粉刷匠，那他们可以分别粉刷一面墙，也就是并行地粉刷两面墙。

探索 Future

Scala 里驱动并发编程的首要工具是 `Future`，位于 `scala.concurrent` 包里。`Future` 可以看作一个容器，里面放了目前还没有生成的值。我们来通过一个简单例子来看看它的用法：

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> import scala.concurrent.ExecutionContext
import scala.concurrent.ExecutionContext

scala> val context: ExecutionContext =
scala.concurrent.ExecutionContext.global
context: scala.concurrent.ExecutionContext =
scala.concurrent.impl.ExecutionContextImpl@3fce8fd9
scala> def example(){

  println("Starting the example")
```

```
  Future{
    println("Starting the Future")
    Thread.sleep(1000)  // simulate computation
    println("Done with the computation")
  }(context)
println("Ending example")
}
```

在这个小方法里，创建了一个 Future 值用来模拟一个费事的运算，程序打印几行信息方便我们理解应用的执行流程。 当运行 example 时，我们看到执行流程如下：

```
scala> example()
Starting the example
Ending example
Starting the future
// a pause
Done with the computation
```

可以看到 Future 在 example 方法结束后执行。这是因为当创建 Future 后，它的运算是并发启动的。你可能会好奇，“创建 Future 时用到的 ExecutionContext 类型的 context 对象是什么东西？”我们很快会深入探讨 ExecutionContext，不过暂时先把它当作一个负责执行 Future 的对象。我们导入了 scala.concurrent.ExecutionContext.global，这是一个标准库里提供的默认对象，可以用来执行 Future。

Future 是一个有状态的对象。它要么处于计算正在执行中的未完成状态，要么处于计算完成状态。更进一步说，一个完成的 Future 要么是 Success，代表计算成功完成；要么是个 Failure，代表在计算过程中抛了异常。

Future API 提供了组合函数（combinator）来组合 Future 实例和对 Future 结果的操作：

```
scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global

scala> import scala.concurrent.Future
import scala.concurrent.Future
```

```
scala> Future(1).map(_ + 1).filter(_ % 2 == 0).foreach(println)

2
```

我们在 Scala 终端里构造了一个 Future 数据类型，里面装了个整数值。接着我们用类似于 Option 和集合类型里的函数来转换里面装的整数值。一旦前序的 Future 完成，里面的转换就会立刻应用，返回一个新的 Future。

前面说过我们会深入看看 ExecutionContext。ExecutionContext 可以理解为 Future 背后的一种提供异步能力的机制。上面的代码创建了一个 Future 来执行加法和取模运算，但是并没有在调用位置显式地提供 ExecutionContext 实例，而是仅仅导入了标准库提供的 global 对象。代码之所以能够运行是因为 global 是个隐式值而 map 的函数签名接受隐式的 ExecutionContext。来看一下下面的 map 方法的签名以加深理解：

```
def map\[S](f: T => S)(implicit executor: ExecutionContext): Future\[S]
```

从 map 的签名可以看出，和 List 的 map 转换方法不同，Future 需要一个柯里化的、隐式的 ExecutionContext 参数。为了理解 ExecutionContext 如何提供异步功能，我们需要先理解它包含哪些操作：

```
trait ExecutionContext {
  def execute(runnable: Runnable): Unit
  def reportFailure(cause: Throwable): Unit
  def prepare(): ExecutionContext = this
}
```

execute 是个对 java.lang.Runnable 做操作的副作用方法。熟悉 Java 并发的开发人员很可能会想起 Runnable 是 Java 里面支持用线程和其他 java.util.concurrent 抽象做异步编程时广为使用的接口。虽然我们还不知道 Future 是怎么做到运行时异步的，从接口上已经可以看出 Future 执行和创建 Runnable 之间存在某种关联。

我们要回答的下一个问题是，“我怎么创建 ExecutionContext？”通过研究其伴生对象，我们发现了如下签名：

```
def fromExecutorService(e: ExecutorService, reporter: Throwable => Unit):
ExecutionContextExecutorService
```

```
def fromExecutorService(e: ExecutorService):
ExecutionContextExecutorService
def fromExecutor(e: Executor, reporter: Throwable => Unit):
ExecutionContextExecutor
def fromExecutor(e: Executor): ExecutionContextExecutor
```

标准库里提供了从 java.util.concurrent.Executor 或 java.util.concurrent.ExecutorService 创建 ExcecutionContext 的方便方法。



如果你不熟悉 java.util.concurrent 包提供的并发机制而想补充一些知识，我们推荐你阅读 Brian Coetz 的 Java Concurrency in Practice（http://jcip.ne t/）。尽管 java concurrency in practice 写于 JDK6 发布前后，它包含很多今天仍然适用的技术。阅读这本书可以让你对 JDK 提供的并发基础机制以及 Scala 标准库如何利用这些机制获得更深的理解。

工厂方法返回的类型是 ExecutionContext 的更特定的版本。标准库为 ExecutionContext 定义了如下的继承层次：

```
trait ExecutionContextExecutor extends ExecutionContext with
java.util.concurrent.Executor
trait ExecutionContextExecutorService extends ExecutionContextExecutor with
java.util.concurrent.ExecutorService
```

我们还在 ExecutionContext 伴生对象里找到了前面例子里用到的隐式值：

```
def global: ExecutionContextExecutor = Implicits.global
```

Implicits.global 的文档指出 ExecutionContext 依赖于一个线程池，线程池里的线程数等于可用核数。对 ExecutionContext 的调研告诉了我们 Future 是怎么运行的。我们来演示一下 Future 是如何用 ExecutionContext 来多线程执行的：

```
Future(1).map(i => {
  println(Thread.currentThread().getName)
  i + 1
}).filter(i => {
  println(Thread.currentThread().getName)
```

```
    i % 2 == 0
  }).foreach(println)
```

我们扩展原始的代码片段让它打印执行转换的线程的名字。当运行在一台多核机器上时，这段代码的输出每次都不一样，依赖于哪个线程被选中来执行转换。下面是例子输出：

```
ForkJoinPool-1-worker-3
ForkJoinPool-1-worker-5
2
```

这个例子显示 worker-3 线程执行 map 转换而另一个 worker-5 线程执行 filter 转换。从这个简单例子可以看出两个影响 Future 控制流的关键认识。首先，Future 是一种用于并发的数据结构，让我们能够把程序的控制流分解成多个逻辑线程处理。第二，例子显示 Future 在创建完成后立即执行。这意味着转换立刻在程序的不同处理流里执行。我们将利用这两点认识来提升 Dave 的 crazy ideas 的运行时性能。

Future 和 crazy ideas

我们应用 Future 来提升 Dave 的回测的性能。相信应该有机会提高性能，因为 Dave 的笔记本有 4 个核。这意味着通过给程序增加并发，我们能够从运行时并行中获益。我们第一次尝试使用 for 推导：

```
implicit val ec = scala.concurrent.ExecutionContext.Implicits.global
for {
     firstDaveAapl <- Future(backtest(Dave1, Ticker("AAPL"),
lastMonths(3)))
     firstDaveGoog <- Future(backtest(Dave1, Ticker("GOOG"),
lastMonths(3)))
     secondDaveAapl <- Future(backtest(Dave2, Ticker("AAPL"),
lastMonths(3)))
     secondDaveGoog <- Future(backtest(Dave2, Ticker("GOOG"),
lastMonths(3)))
   } yield (firstDaveAapl, firstDaveGoog, secondDaveAapl, secondDaveGoog)
```

每个回测调用都被包装在用 Future.apply 创建的 Future 实例里。这个伴生对象方法用 by-name 参数来推迟参数的运算，在这个例子里也就是对回测的调用：

```
def apply[T](body: =>T)(executor: ExecutionContext): Future[T]
```

运行新版 CrazyIdeas.scala 之后，你沮丧地发现运行时性能并未提高。你快速地复查了一下你的 Linux 机器上的 CPU 核数：

```
$ cat /proc/cpuinfo  | grep processor | wc -l
8
```

确认你的笔记本电脑上确实有 8 个核之后，你很奇怪为什么执行时间和前面串线的时候一样。你需要反思一下 for 推导会编译成什么样子。for 表达式等价于下面这个简单的例子：

```
Future(1).flatMap(f1 => Future(2).flatMap(f2 => Future(3).map(f3 => (f1,
f2, f3))))
```

从这个去糖后的 for 推导，可以看到 Future 的第二个参数是在第一个 Future 的 flatMap 里创建和运算的。应用于 Future 的任何转换函数（比如说 flatMap）只有在转换的参数计算出来后才会执行。这意味着上例中的 Future 和 for 推导是按顺序执行的。为了达成并发效果，我们必须把 CrazyIdeas.scala 修改成下面这样：

```
val firstDaveAaplF = Future(backtest(Dave1, Ticker("AAPL"),
  lastMonths(3)))
val firstDaveGoogF = Future(backtest(Dave1, Ticker("GOOG"),
  lastMonths(3)))
val secondDaveAaplF = Future(backtest(Dave2, Ticker("AAPL"),
  lastMonths(3)))
val secondDaveGoogF = Future(backtest(Dave2, Ticker("GOOG"),
  lastMonths(3)))
for {
  firstDaveAapl <- firstDaveAaplF
  firstDaveGoog <- firstDaveGoogF
  secondDaveAapl <- secondDaveAaplF
  secondDaveGoog <- secondDaveGoogF
} yield (firstDaveAapl, firstDaveGoog, secondDaveAapl, secondDaveGoog)
```

这段代码里，4 个回测同时并发发动，结果被转换为一个包含四元组的 Future 来存放 4 个 BacktestPerformanceSummary 值。眼见为实，在给 Dave 演示了他的回测现在跑得多快后，他兴奋地翻看着新的回测结果。Dave 说了句双关语，“把所有核都用上让我的笔记本风扇（fans）嗖嗖地转。我不确定是不是噪声的粉丝（fans），不过我确定爱死这性能了！”

Future 使用时的考量

在前面例子中，我们通过为回测引入并发给大家展示了 Future API 的易用性。和任何强大的工具一样，使用 Future 必须遵守一些纪律来确保正确性和性能。本节讨论一些在引入 Future 时常遇见的困扰和错误。我们将详细讨论执行副作用，阻塞执行，处理错误，选择合适的 execution context 和性能考虑。

执行副作用

当用 Future 编程时，很重要的是记住 Future 天然是个副作用构造。除了用 success 或 failure 工厂方法来把一个值提升到（lift）Future 里这种场景外，其他情况下工作都是排期到（由创建 Future 的 ExecutionContext 管理的）别的线程里执行的。更重要的是，一旦执行过了，Future 就不能再被重复执行。考虑下面代码片段：

```
scala> import scala.concurrent.Future
import scala.concurrent.Future
scala> import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.ExecutionContext.Implicits.global
scala> val f = Future{ println("FOO"); 40 + 2}
FOO
f: scala.concurrent.Future[Int] =
scala.concurrent.impl.Promise$DefaultPromise@5575e0df
scala> f.value
res3: Option[scala.util.Try[Int]] = Some(Success(42))
```

Future 被计算并且符合期望地打印了 FOO。然后我们可以再次访问 Future 里包装的值，但是注意在访问值的时候，终端没有任何打印输出。一旦完成，Future 就只不过是一个实际值的包装了。如果你想再次执行运算，你需要构造一个新的 Future。



前面的例子用 Future.value 提取计算的结果。这纯粹是出于简化的目的。生产代码应该极少使用这个方法。其返回类型定义为 Option[Try[A]]。选择 Option 是为了用 Some 来代表完成的 Future，用 None 来代表没有完成的 Future。进一步看，记住完成的 Future 有两种状态：成功或失败。里面那个 Try 就用于这个目的。和 Option.get 一样，用 Future.value 绝大部分

情况下都不是好主意。要提取 Future 里的值，请参考后面讲述的其他技巧。

阻塞执行

在给回测添加并发功能时，我们写了个 for 推导，返回 Future[(BacktestPerformanceSummary, BacktestPerformanceSummary, BacktestPerformanceSummary, BacktestPerformanceSummary)]，这可能让你奇怪，你要怎么访问包装在 Future 里的值呢？另一种提问的方式是，“给定 Future[T]，我怎么返回 T？”简单的答案是“不需要！”用 Future 编程需要做一个从同步执行到异步执行的思维方式转换。在做异步编程时，目标是避免直接对 T 做处理，因为这意味着同步。

在实践中，存在 Future[T] => T 会有用的一些场景。比如，考虑回测代码。如果代码用于构造一个扩展 App 的程序，这个程序就会在回测完成前先退出。因为 ExecutionContext global 里的线程还是守护线程，JVM 会在创建完 Future 后立刻退出。在这种情况下，我们需要某种同步机制来暂停执行指导测试结果完成。通过扩展 Awaitable 特质，Future 可以提供这种机制。Await 模块暴露两个方法来达成这个目的：

```
def ready[T](awaitable: Awaitable[T], atMost: Duration): awaitable.type
def result[T](awaitable: Awaitable[T], atMost: Duration): T
```

由于 Future 扩展 Awaitable，Future 可以作为参数传递给这两个方法。ready 挂起当前程序流，指导 T 可用并返回完成 Future[T]。在实践中，ready 很少使用，因为一个同步调用返回 Future[T]而不是 T，在概念上有点费解。更常用的是 result，提供了期望的从 Future[T]到 T 的转换。比如说，CrazyIdeas.scala 可以修改成这样：

```
    val summariesF = for {
      firstDaveAapl <- Future(backtest(Dave1, Ticker("AAPL"),
lastMonths(3)))
      firstDaveGoog <- Future(backtest(Dave1, Ticker("GOOG"),
lastMonths(3)))
      secondDaveAapl <- Future(backtest(Dave2, Ticker("AAPL"),
lastMonths(3)))
      secondDaveGoog <- Future(backtest(Dave2, Ticker("GOOG"),
lastMonths(3)))
```

```
    } yield (firstDaveAapl, firstDaveGoog, secondDaveAapl, secondDaveGoog)

    Await.result(summariesF, scala.concurrent.duration.Duration(1,
  java.util.concurrent.TimeUnit.SECONDS))
```

在这段代码里，可以看到用阻塞、同步的 Await.result 调动来返回 Future[BacktestPerformanceSummary]元组。这个阻塞调用接受一个超时参数，用来防范 Future 计算不能在指定时间内完成的场景。用阻塞调用来获取回测结果意味着 JVM 只有在回测完成或者超时时才会退出。如果超时时间到而回测没有完成，result 和 ready 会抛出 TimeoutException。

阻塞执行对程序性能有潜在危害，应该慎用。用 Await 伴生对象的方法使阻塞调用更容易识别。因为 ready 和 result 在超时时抛出异常，你应该更谨慎地处理而不是返回不同的结果类型。你应该把任何涉及异步的同步调用（却没有提供超时或处理超时错误）视为 bug。

异步编程需要转换思维方式为描述当将来的值出现时要怎么做而不是写需要值已经存在然后去处理的程序。你应该怀疑任何使用 Await 来打断对将来的值的转换这种写法。应该基于 Future[T]做一系列转换而不是基于 T。对于 Await 的使用应该限制在程序没有别的事要做，只差转换的结果的时候，就像前面的回测程序那样。



由于标准库用异常而不是不同的返回类型来建模超时，因此难以确保超时是否被正确处理了。提高安全性的一种方法是写一个工具方法，返回 Option[T]而不是 T 来应对超时场景：

```
object SafeAwait {
  def result[T](
    awaitable: Awaitable[T],
    atMost: Duration): Option[T] =
    Try(Await.result(awaitable, atMost)) match {
      case Success(t) => Some(t)
      case Failure(_: TimeoutException) => None
      case Failure(e) => throw e
    }
```

用这个新方法，一整类的错误被消除了。以后你遇到别的不安全转换时，考虑定义返回把错误场景考虑在内的数据类型，以免对转换结果处理不正确。你想到了什么别的不安

全转换呢?

处理错误

使用 Future 需要有纪律地处理错误场景以避免写出一组难以理解的转换。当一个 Future 转换里抛出异常时，它在转换操作的线程里出现并打断下游的转换。考虑下面这个例子：

```
Future("not-an-integer").map(_.toInt).map(i => {
     println("Multiplying")
     i * 2
   })
```

你预期在第一个 map 转换后会发生什么？很明显转换会失败，因为提供的参数无法转换成整型。在这种场景下，这个 Future 被视为失败，而后续的对包装的 Int 值做转换的操作，在这个例子里，都不会发生。在这个简单例子里，转换显然无法继续。想想在一个大代码库里，操作比简单的整数复杂得多的数据，存在多种失败场景，跨越多个命名空间和源代码文件。在真实世界里，识别一个异步操作会不会挂掉远不是这么简单的。

Future 提供了机制来处理失败。它提供了 recover 和 recoverWith 来继续下游的转换。方法签名如下：

```
def recover[U >: T](pf: PartialFunction[Throwable, U])(implicit executor:
ExecutionContext): Future[U]
def recoverWith[U >: T](pf: PartialFunction[Throwable, Future[U]])(implicit
executor: ExecutionContext): Future[U]
```

这两种恢复方法的区别在于提供给 recover 的偏函数返回 U，而提供给 recoverWith 的偏函数返回 Future[U]。在前例中，我们可以用 recover 来提供一个默认值以便继续转换，如下：

```
Future("not-an-integer").map(_.toInt).recover {
  case _: NumberFormatException => -2
}.map(i => {
  println("Multiplying")
  i * 2
})
```

运行这段代码产生如下输出：

```
Multiplying
Multiplication result = -4
```

这种手段让你能够在一个转换失败的时候后续地转换，但是存在和 Await 方法一样的缺陷。返回的 Future[T] 数据类型不能反映可能的失败。用 recovery 方法很容易出错，因为只有通过阅读代码才能知道错误条件是否处理了。

我们目前学到的错误处理都适用于处理计算中的失败。很可能在一系列转换后，你希望执行某种特别逻辑。假设你在构造一个 web service 向股票交易所提交交易指令。只有当指令成功提交给交易所后才认为指令提交成功，其他情况下都视为提交失败。由于指令提交涉及与外部系统——交易所——的通信，你把它建模为 Future。下面是提交指令方法的代码：

```
def submitOrder(
  ec: ExecutionContext,
  sendToExchange: ValidatedOrder => Future[OrderSubmitted],
  updatePositions: OrderSubmitted => Future[AccountPositions],
  o: RawOrder): Unit = {
  implicit val iec = ec
  (for {
    vo <- ValidatedOrder.fromRawOrder(o).fold(
      Future.failed[ValidatedOrder](new Exception(
      "Order failed validation")))(Future.successful)
    os <- sendToExchange(vo)
    ap <- updatePositions(os)
  } yield (os, ap)).onComplete {
    case Success((os, ap)) => // Marshal order submission info to caller
    case Failure(e) =>  // Marshal appropriate error response to caller
  }
}
```

一个 ExecutionContext，一种提交指令的方法和一种在指令提交成功后更新客户交易位置的方法允许客户提供要提交给交易所的 RawOrder。在第一个处理步骤中，RawOrder 被转换成 ValidatedOrder，然后提升（lift）到 Future 里。Future.failure 和 Future.successful 是把一个值提升或包装成 Future 的方便方法。值被提升到

Future 里，以便让整个一系列操作可以写成一个 for 推导。

随着所有处理步骤的完成，onComplete 被调用来异步地完成请求过程。你可以想象在这个上下文里，完成请求过程意味着响应结果的序列化和传送给调用端。之前我们只介绍了一种在值计算出来的时候执行某个操作的方式，就是用 Await。onComplete 注册一段回调函数，当 Future 值完成的时候被异步调用。如例子中所示，onComplete 支持成功场景和失败场景的处理，是用一种全功能的工具来处理 Future 转换的结果。除了 onComplete 外，Future 还专门为失败场景提供了 onFailure，为成功场景提供了 onSuccess 和 foreach。

这些回调方法的暴露的方法签名返回 Unit。作为一个函数式程序员，你应该对调用这类方法持有猜疑态度，因为它们是有副作用的。onComplete 应该仅在实际计算链的最后才调用，因为这时候已经无法再进一步推迟副作用了。以 web service 例子来说，副作用是把响应结果传送给调用端。另一个使用这种副作用回调的常用场景是在处理横切关注点（cross-cutting concern）时，比如应用度量。回到 web service 例子，下面代码在向交易所提交指令失败时把出错次数加 1：

```
(for {
  vo <- ValidatedOrder.fromRawOrder(o).fold(
    Future.failed[ValidatedOrder](
    new Exception("Order failed validation")))(Future.successful)
  os <- {
    val f = sendToExchange(vo)
    f.onFailure({ case e => incrementExchangeErrorCount() })
    f
  }
  ap <- updatePositions(os)
} yield (os, ap))
```

在这段代码里，当提交交易指令失败时通过 onFailure 回调执行副作用操作。在这个例子里，跟踪副作用发生的位置很直观。但是，在一个大系统里定位回调注册的时间和地点是不容易的事。另外，根据 Future API 文档的描述，我们了解到回调的执行是无序的，也就是说所有的回调都要独立看待。这就是为什么你必须有纪律地选择使用副作用的时间和地点。

另一种处理错误的办法是用一种能包装错误的数据类型。我们已经在讲 Await 的时候

看到可以用 Option 来返回结果。用 Option 明示了该计算可能出错，同时保持了易用性，因为 Option 提供了对它包装的值的方便的转换函数（比如说 map）。不幸的是 Option 不允许我们包装错误信息。在这种情况下，来自 Scalaz 库的另一个工具就有用了，这个工具叫作 disjunction。Disjunction 类似于 Either，可以用来代表两种可能类型中的一个。与 Either 不同的是 Disjunction 的操作是偏右的（right-biased）。我们来看个简单例子：

```
scalaz.\/.right[Throwable, Int](1).map(_ * 2)
```

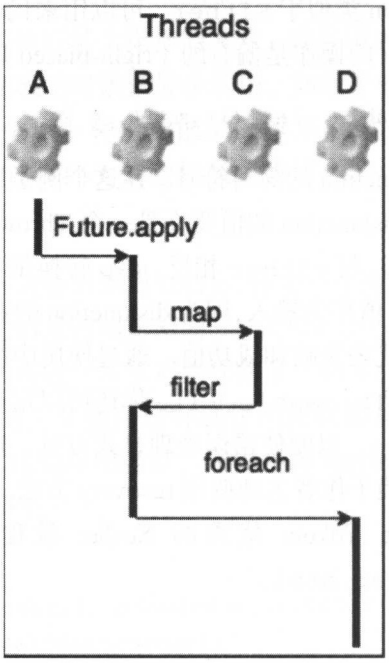
\/是 Scalaz 里代表 disjunction 的简写符号。在这个例子里通过包装一个整数常量创建了一个右 disjunction。这个 disjunction 的值要么是一个 Throwable 要么是个 Int 值，类似于 Either[Throwable, Int]。与 Either 相反，map 转换是对 disjunction 的右值执行的。在这个例子里，map 接受 Int 值作为输入，因为 disjunction 的右侧值是 Int。因为 disjunction 是偏右的，它天然适合用来代表失败和成功值。通过使用中缀操作符，通常把带错误处理的 Future 定义为 Future[Throwable \/ T]。你还可以定义一个包含所有可能的出错类型的 ADT 来替代 Throwable，以便使错误处理显式可见。这个方法很不错，因为它可以强制处理错误场景而无须依赖于作者主动调用 recovery 方法。如果你想学习关于 disjunction 的更多知识，参考 Eugene Yokota 优秀的 Scalaz 教程 http://eed3si9n.com/learning-sc alaz/Either.html。

提交执行妨碍性能

由于 Future 提供了表达力强又易于使用的 API，在大系统里完成计算前经常会做大量的转换（transform）。回忆上节的指令提交 web service 例子，你可以想到多个应用层在操作一个 Future。用于生产的 web service 通常组合多个层次的计算来伺服一个请求。一个请求处理流程可能包含以下阶段：请求的反序列化，鉴权，调用应用服务，数据库查询或调用第三方服务，然后把响应结果转换成 JSON 格式。如果这个流程中的每个阶段都建模为 Future，那么通常会有 5 个或更多转换来处理一个请求。

用这种方式把你的软件系统分解成多个职责不同的小块是一种好的工程实践，有利于分别测试和提升可维护性。然而这样使用 Future 来设计提升的同时伴随着性能成本。如我们在例子中看到的，几乎所有基于 Future 的转换都需要向 Executor 提交工作。在上述的工作流程中，大多数的转换都很小。在这种场景下，把工作提交给 executor 的开销占了执行时间的主要部分。如果指令提交 web service 要服务大量的客户同时有严格的吞吐量和时延要求，那么有可能可测试性和可维护性方面的工程时间会导致差的性能表现。

如果你研究一下这幅图，你可以看到一个带有 4 个线程的线程池用来对 Future 应用转换。每个转化都被提交给线程池，有可能为每个计算选择不同的线程。从这张图可以看得很清楚，由于提交 Executor 的成本，多个小的转换会妨碍性能。



提交 Executor 的成本具体有多大呢？这个问题驱使我们写一个基准测试来量化其开销。这个基准测试专注于对一个整数加 1，用两种方式执行 *N* 次。一种方式是在单独一个 Future 里执行加法，第二种方式是每次加法都在一个新的 Future 转换里执行。第二种方式模拟指令提交处理流程例子类似在大系统里用多个 Future 转换的情形。加法用来模拟业务操作，由于它的成本极低，因此可以凸显提交执行的成本。基准测试代码如下：

```
@Benchmark
def manyTransforms(state: TransformFutureState): Int = {
  import scala.concurrent.ExecutionContext.Implicits._
  val init = Future(0)
  val res = (1 until state.operations).foldLeft(init)
    ((f, _) => f.map(_ + 1))
  Await.result(res, Duration("5 minutes"))
}
@Benchmark
```

```
def oneTransform(state: TransformFutureState): Int = {
  import scala.concurrent.ExecutionContext.Implicits._
  val res = Future {
    (1 until state.operations).foldLeft(0)((acc, _) => acc + 1)
  }
  Await.result(res, Duration("5 minutes"))
}
```

TransformFutureState 使操作次数可以参数化。manyTransforms 代表每次用 map 转换来做加法的方式，这种方式会造成提交 Executor 的成本。oneTransform 通过 Future.apply 来做，所有加法只提交一次。在这个受控的测试里，用 Await.result 来作为等待计算结束的阻塞机制。在一台双核机器上跑这个测试来做 5 次和 10 次转换的结果如下：

基准测试	次　数	吞吐量（ops）	错误率（占吞吐量百分比）
manyTransforms	5	463,614.88	± 1.10
oneTransform	5	412,675.70	± 0.81
manyTransforms	10	118,743.55	± 2.34
oneTransform	10	316,175.79	± 1.79

对于 5 次转换的测试，两种方式的性能接近，而 10 次转换的测试就产生明显差异了。这个基准测试清楚表明提交 Executor 的成本可能影响很大。虽然成本可能会高，但是我们给你的建议在前期系统建模时不要考虑这个成本。以我们经验看，对建模良好的系统做改造来提升系统要比改造一个性能很好但是建模很差的系统要容易得多。因此，我们不建议复杂系统一开始就尝试把多个 Executor 提交合并到一起来提升性能。

一旦你有了良好设计，第一步是通过基准测试 profile 来识别 Executor 提交是不是瓶颈。如果你发现你的 Future 使用风格造成了性能瓶颈，你可以考虑几种改善途径。

最低成本的选项是用 Future.success 或 Future.failure 来替换不必要的（成本高昂的）Future 创建。指令提交 web service 利用这些工厂方法来把值提升为 Future。由于值是已经计算好的，这些工厂方法避免了向 Executor 提交任务。在值已经计算好的情况下，用 Future.successful 或 Future.failure 替换 Future.apply 可以降低开销。

另一种开发成本较大的办法是重写实现来把多个类似 manyTransform 的 Future 转

换归并起来。这种战术需要复查每个应用层，判断单独层的转换是否可以组合。如果可以，我们建议你避免跨应用层合并（比如在请求反序列化或鉴权和应用服务处理之间），因为这会削弱你的模型而增加维护成本。

如果两种方法都不解决问题，那么也许值得和产品所有者讨论通过硬件来解决问题。因为你的系统设计没有打折扣，采用了可靠的工程实践，很可能可以水平伸缩或者集群化。根据你系统的状态跟踪方式，有可能无须开发工作就能实现这个选项。也许产品所有者对维护性和扩展性比对性能更重视。如果这样，给系统增加伸缩性可能是可行的方案。

如果你实在没法绕过性能要求，那么还有另外 3 种可能办法。一种选择是调研 `Future` 的一个替代品，叫作 `Task`。Scalaz 库提供的这个控制块，让计算可以用较少的 `Executor` 提交来执行。这个选择开发成本很高，因为所有 `Future` 数据类型都需要替换成 `Task`。我们将在本章末尾探讨 `Task`，调研它能提供的性能收益。

除了使用 `Task` 外，也许有必要复查一下应用的模型，仔细查看关键路径上是否存在不必要的工作。如我们在 MV 交易公司的基础设施和流式处理介绍中所看到的，有时候可以通过重新思考设计来提高性能。和引入 `Task` 一样，重新设计系统架构是很大规模的修改。

最后一种解决方法是合并不同层的 `Future` 来归并 `Future` 转换。我们不建议采取这种办法，除非所有的其他建议都不奏效。这种做法导致难以理解的代码库，因为不同的关注点被混杂了。你可能会获取短期的性能提升，但根据我们的经验，性能收益会被长期的维护和扩展成本抵消掉。

处理阻塞调用和回调

如本章第一部分所述，`Future` API 提供了一种优雅的方式来写并发程序。由于在 `Future` 上阻塞被视为一种不好的实践，很容易看到在整个代码库里到处使用 Future。但是，你的系统不太可能只由你自己的代码组成。大部分真实应用利用已有的库和第三方软件来避免为一些通用问题重新实现已经存在的解决方案（比如数据编码解码，HTTP 通信，数据库驱动，等等）。不幸的是，不是所有的库都用 `Future` API，因此把这些库优雅地集成到应用里来可能变成一个挑战。在本节里，我们会检查你可能会碰见的几个常见陷阱并提出可能的解决方案。

ExecutionContext 和阻塞调用

在处理回测的时候，你注意到代码的一个模块是用来从关系型数据库里加载一些购买指令的历史数据的。在你重写它以利用 Future 的好处的过程中，你把这个模块的 API 完全异步化了。

```
def findBuyOrders(
 client: ClientId,
 ticker: Ticker)(implicit ec: ExecutionContext): Future[List[Order]] = ???
```

但是在测试了应用后，你注意到这部分代码性能相当差。你尝试增加数据库连接数，先两倍，再三倍，但是都不奏效。为了理解问题的根源，你查看了这个方法的所有调用点，然后你发现了如下的模式：

```
import scala.concurrent.ExecutionContext.Implicits.global
findBuyOrders(clientId, tickerFoo)
```

所有的调用方都调入了全局的 ExecutionContext 并被这个方法隐式地使用。默认连接池基于 ForkJoinPool，其大小基于机器的可用核数。因此，它是 CPU 受限（CPU-bound）的，设计来处理非阻塞的，CPU 密集的操作。对于执行非阻塞调用的应用来说，这是个好的选择。但是你的应用异步的运行阻塞调用（在 Future 里），这种场景下使用默认的 ExecutionContext 很可能导致性能快速下降。

异步 vs 非阻塞

在继续讲解之前，我们想澄清本节中使用的几个术语。非阻塞在并发的上下文下可能是一个让人困惑的术语。当使用 Future 时，我们执行了一个异步操作，意味着启动了一个可以和当前程序流一起执行的运算。这个运算在后台执行，最终会产生一个结果。这种行为有时候被称作非阻塞，意思是 API 调用立刻返回了。然而，阻塞和非阻塞更经常用来指 I/O 操作是如何执行的，尤其是执行操作的线程是怎么使用的。比如说，把字节序列写入本地文件可能是个阻塞操作，因为调用 write 的线程不得不等待（阻塞），直到 I/O 操作完成。如果使用的是非阻塞构造，比如 java.nio 包提供的那些，执行 I/O 操作可能不必阻塞线程。

实现一个组合了如下行为的 API 是可能的：

API 特点	返　回	阻塞线程?
同步/阻塞	计算完成时	是的，执行操作的线程
异步/阻塞	立刻	是的，阻塞线程池里的某个线程
异步/非阻塞	立刻	不，当执行阻塞操作时线程被释放

为阻塞调用使用专门的 ExecutionContext

显然，问题出来我们用全局的 ExecutionContext 来执行阻塞调用。我们在查询关系型数据库，而大部分 JDBC 驱动都是以阻塞调用的方式实现的。线程池里的线程调用 JDBC 驱动然后阻塞等待查询结果通过网络传回来，因此无法被用于别的计算。一种解决方法是创建一个专门用来执行阻塞时 Future 的 ExecutionContext。这个 ExecutionContext 线程池配置较多的线程数，因为已经预期它会用来执行阻塞调用：

```
val context = ExecutionContext.fromExecutorService(
  Executors.newFixedThreadPool(20)
)
findBuyOrders(clientId, tickerFoo)(context)
```

第一个好处是现在我们的线程数变多了，因此可以发起更多的并发查询。第二个好处是系统的其他异步调用在独立的线程池（比如 global）里运行，它们不会被阻塞线程所影响。

我们写了个小基准测试来评估新实现的性能。在这个例子里，我们用一个假的 findBuyOrders 实现来模拟查询数据库：

```
def findBuyOrders(
 client: ClientId,
 ticker: Ticker)(ec: ExecutionContext): Future[List[Order]] = Future {
 Thread.sleep(100)
 Order.staticList.filter(o => o.clientId == client
   && o.ticker == ticker)
}(ec)
```

我们把 ExecutionContext 作为参数传进去。基准测试对比了两种实现方式：一种用默认的 ExecutionContext，另一种用配置了 20 个线程的阻塞调用专用 ExecutionContext 实现，结果如下：

基准测试	操作数	吞吐量（ops）	错误率（占吞吐量百分比）
withDefaultContext	10	3.21	± 0.65
withDedicatedContext	10	9.34	± 1.00
withDefaultContext	1,000	0.04	± 2.56
withDedicatedContext	1,000	0.73	± 0.41

结果验证了我们的直觉。专用线程池考虑到了会有线程阻塞等待操作完成，所以配置的线程数要比默认的大。因为有了更多的可用线程数，所以能够并发启动更多的阻塞操作，这样就取得了更好的吞吐量。创建一个专用的 ExecutionContext 是一种好的方式来隔离阻塞操作，确保阻塞操作不会减慢 CPU 密集任务的运算。在设计专用线程池时，确保你理解了底层的资源是怎么被使用的（比如说，数据库连接，文件句柄，等等）。比如说，在处理关系型数据库时，我们知道一个连接同一时间只能执行一个查询。有一个不错的经验法则是让线程池的线程数等同于你将要打开的数据库连接数。如果数据库连接数小于线程数，有些线程可能要等待连接而无法使用。如果数据库连接数大于线程数，则情况相反，可能有些数据库连接会闲置。

有一种好的策略是依赖于类型系统和编译器来确保你不会搞混不同的 ExecutionContext 实例。除非类型不同，不然你有可能会在执行阻塞操作时误用 CPU 密集的 ExecutionContext。你可以自定义一个 DatabaseOperations ExecutionContext 来包装一个 ExecutionContext，然后在构造数据库访问模块的时候使用这个自定义类型。另一个主意是用 Salaz 提供的标记类型。参考第 3 章来重温一下标记类型。考虑如下例子：

```
object DatabaseAccess {

 sealed trait BlockingExecutionContextTag

 type BlockingExecutionContext = ExecutionContext @@
BlockingExecutionContextTag

 object BlockingExecutionContext {
  def fromContext(ec: ExecutionContext): BlockingExecutionContext =
    Tag[ExecutionContext, BlockingExecutionContextTag](ec)

 def withSize(size: Int): BlockingExecutionContext =
```

```
fromContext(ExecutionContext.fromExecutor(Executors.newFixedThreadPool(size
)))
}
}

class DatabaseAccess(ec: BlockingExecutionContext) {
  // Implementation elided
}
```

为 ExecutionContext 应用 tagged 类型给我们增加了安全性。在开发你自己的应用的时候很容易在 main 方法里犯错误，而且在创建你自己的模块时也容易无意中用了错误的 ExecutionContext。

使用 blocking 语句

（译者注，blocking construct 严格来说不是语句，而是用函数模拟出来的控制结构，但是在下面的例子和这个上下文里翻译成语句更通顺一点。）

标准库提供了 blocking 语句用来标示在 Future 里面执行的阻塞操作。我们可以修改前面的例子，使用 blocking 而不是专用 ExecutionContext：

```
import scala.concurrent.ExecutionContext.Implicits.global
def findBuyOrders(
 client: ClientId,
 ticket: Ticker): Future[List[Order]] = Future {
   scala.concurrent.blocking{
     Thread.sleep(100)
     Order.staticList.filter(o => o.clientId == client && o.ticker ==
ticker)
   }
}
```

注意在前面的实现里，我们用 ExecutionContext 来执行 Future。blocking 语句用来通知 ExecutionContext 一个计算是阻塞式的。这使 ExecutionContext 能够适配其执行策略。比如说，全局 ExecutionContext 会在执行包装在 blocking 语句里的计算时临时增加线程数。在线程池里为阻塞语句创建一个专用线程，以便确保其他线程仍然用于 CPU 密集型计算。

你应该谨慎使用 blocking。blocking 仅仅用于通知 ExecutionContext 被包装的操作是阻塞的。是实现某种特定的策略还是忽略通知是 ExecutionContext 的职责。只有默认的全局 ExecutionContext 考虑了这种情况并提供了特殊策略。

用 Promise 转化回调

除了 Future 是 scala.concurrent API 的主要控制结构外，还有另一个有用的抽象叫作 Promise。Promise 是另一种创建和完成 Future 的方式。Future 是个只读容器，里面放了最终会被计算出来的值。Promise 是个句柄，让你能够现实地设置 Future 里的值。一个 Promise 永远只跟一个 Future 关联，反之亦然。可以把这个 Promise 所属的 Future 的用成功结果来完成或抛异常（将导致 Future 失败）。

我们来看个例子以便理解 Promise 是怎么工作的：

```
scala> val p = Promise[Int]  // this promise will provide an Int
p: scala.concurrent.Promise[Int] =
scala.concurrent.impl.Promise$DefaultPromise@d343a81

scala> p.future.value
res3: Option[scala.util.Try[Int]] = None
// The future associated to this Promise is not yet completed

scala> p.success(42)
res4: p.type = scala.concurrent.impl.Promise$DefaultPromise@d343a81

scala> p.future.value
res5: Option[scala.util.Try[Int]] = Some(Success(42))
```

一个 Promise 只能使用一次来完成它所关联的 Future，要么成功，要么失败。尝试完成一个已经完成的 Promise 会抛异常，除非你用 trySuccess、tryFailure 或 tryComplete。这 3 个方法会尝试完成 Promise 所关联的 Future，如果成功完成则返回 true，如果之前已经完成过了则返回 false。

看到这里，你可能会疑惑到底什么场合下使用 Promise 会有好处。特别是对于前面的例子，简单地返回内部的 Future 而不是使用 Promise 是不是更简单一些？记住前面的例子只是用来展示 Promise 的简单使用流程。不过，你的问题可以理解。在实践上，

我们看到两种使用 Promise 的常见场景。

从回调到基于 Future 的 API

第一种场景是把一个基于回调的 API 变成基于 Future 的 API。假设你要集成一个第三方产品，比如 MV 交易公司最近刚刚买了使用许可的数据库。这个数据库用来存储基于时间戳的历史数据非常棒。它还提供了供客户端使用的库。不幸的是，这个库虽然是全异步非阻塞的，但是是基于回调的：

```
object DatabaseClient {
  def findQuote(instant: Instant, ticker: Ticker,
    f: (Quote) => Unit): Unit = ???

  def findAllQuotes(from: Instant, to: Instant, ticker: Ticker,
    f: (List[Quote]) => Unit, h: Exception => Unit): Unit = ???
}
```

这个客户端无疑能很好地工作；毕竟 MV 交易公司为它花了不少钱！

但是把它和你自己的应用集成起来不太容易。你的程序大量依赖于 Future。这正是 Promise 可以发挥作用的地方：

```
object DatabaseAdapter {

  def findQuote(instant: Instant, ticker: Ticker): Future[Quote] = {
    val result = Promise[Quote]

    DatabaseClient.findQuote(instant, ticker, {
      q: Quote =>
        result.success(q)
    })

    result.future
  }

  def findAllQuotes(from: Instant, to: Instant, ticker: Ticker):
  Future[List[Quote]] = {
    Val result = Promise[List[Quote]]
```

```
      DatabaseClient.findQuote(from, to, ticker, {
        quotes: List[Quote] => result.success(quotes)
      }, {
        ex: Exception => result.failure(ex)
      }
    }
    result.future
  }
```

幸亏有 Promise 抽象，我们可以返回一个 Future。我们简单地在相应的回调里用 success 和 failure 来调用数据库客户端。这种情况在集成 Java 库的时候经常发生。尽管 Java 8 对 Java concurrent 包做了大量改进，大部分 Java 库还是依赖于用回调来实现异步。使用 Promise，你可以完成发挥已有 Java 生态圈的能力而无须放弃 Scala 提供的异步编程优势。

组合 Future 和 Promise

Promise 还能用来组合 Future 实例。比如说，我们给 Future 加上超时功能：

```
def run[A](f: => Future[A], timeout: Duration): Future[A] = {
  val res = Promise[A]
  Future {
    Thread.sleep(timeout.getMillis)
     res.tryFailure(new Exception("Timed out")

  }
  f onComplete {
  case r => res.tryCompleteWith(f)
  }

  res.future
}
```

我们的方法是接受一个 by-name 的 Future（也就是说，Future 还没有开始启动执行）和一个超时时间。在方法里，用一个 Promise 作为结果的容器。启动一个内部 Future，这个 Future 会阻塞直到超时，然后用一个异常使 Promise 失败。同时启动主要的 Future 并注册一个回调，用计算结果来完成 Promise。注意在这个例子里，我们用 tryFailure

和 tryCompleteWith。很可能两个 Future 最终都会终结并试图完成 Promise。我们只对先完成的 Future 的结果感兴趣，但同时也要避免在试图完成一个已经完成的 Promise 时抛异常。



前面的例子是一个过于简单设计的超时。最多只能算作一个原型，用来演示怎样利用 Promise 来增强 Future 和实现复杂的功能。更实际的实现很可能设计 ScheduledExecutorService。ScheduledExecutorService 允许你把安排在一段时延后再执行运算。它使我们可以排期调用 tryFailure 而不用调用 Thread.sleep 来阻塞线程。为了保持例子简单，我们决定不引入新类型，但是建议你研究 ScheduledExecutorService 的实现。

在实践中，你可能有时候不得不为 Future 写自定义的组合子（combinator）。Promise 是一个有用的工具帮你做到这一点。不过，Future 及其伴生对象已经提供了大量的内建组合子，你应该优先使用内建的功能。

受命进一步提升回测性能

发现 Future 并且采用异步思维方式帮助你更好地利用计算资源来更快地测试多种交易策略。你把回测当作黑盒来提高性能，不用修改回测的实现就取得了明显的性能提升。识别能够并行化的串行工作时考虑提升性能方法时的好思路。

让我们在回测内部的一个较小的逻辑处理流程进一步把这种思路发扬光大。回测对跨越一段时间周期的时点应用交易策略（A backtest exercises a strategy for a ticker across a time period）。在跟 Dave 讨论后，你发现 MV 交易公司不把持仓（maintain position）过夜。每个交易日结束时，MV 交易公司的交易系统通过确保所有仓位都已清偿来减小风险。这是为了应对股市关闭后的夜间价格波动，公司无法对这样的波动做出反应。因为不持仓过夜，每个交易日都可以独立于别的前个交易日来模拟。回到我们的异步思维方式，这个发现意味着交易日模拟可以并发执行。

在立刻去用 Future 实现之前，我们想分享另一种由 Scalaz 库提供的抽象，叫作 Task。有很好的理由让我们建议用 Task 来修改回测。

介绍 Scalaz Task

Scalaz Task 提供了另一种达成并发的方式。虽然 Task 可以按照类似的 Future 的方式来使用，但是它们之间存在重要的概念差别。Task 允许对异步执行的细粒度的控制，这能带来性能收益。Task 还保持了引用透明性，使代码更容易推理。引用透明是无副作用的表达式的一个好处。为了更好地理解其原理，考虑如下简单的 sum 方法：

```
def sum(x: Int, y: Int): Int = x + y
```

假设我们要执行两个求和：

```
sum(sum(2, 3), 4)
```

因为 sum 是无副作用的，所以可以用 sum(2,3) 的结果替换它，变成这样：

```
sum(5, 4)
```

这个表达式的结果永远是 9，符合引用透明的要求。现在假设修改一下 sum 的实现：

```
class SumService(updateDatabase: () => Unit) {
  def sum(x: Int, y: Int): Int = {
    updateDatabase()
    x + y
  }
}
```

现在 sum 包含了一个写入数据库的副作用，这破坏了引用透明性。我们不再能够把 sum(2,3) 替换成值 5 了，因为这样就不会更新数据库了。引用透明是函数式编程范式的核心概念之一，因为它使代码容易推理。Haskell wiki 提供了更多的补充和例子，值得一读：https://wiki.haskell.org/Referential_transparency。

让我们来看看常用的 Task API 的用法来更好地理解 Task 是怎么工作的。

Task 的创建和执行

Task 伴生对象提供的方法是 API 的主要入口，也是创建 Task 实例的最好方式。Task.apply 是第一个要检查的方法。它接受一个返回 A 实例的计算（实际上是一个 A 类型的 by-name 参数）和一个隐式的 ExecutorService 来运行计算。和用 ExecutionContext 作为线程池抽象的 Future 不同，Task 使用 Java 标准库里定义的

ExecutorService:

```
scala> val t = Task {
     |   println("Starting task")
     |   40 + 2
     | }
t: scalaz.concurrent.Task[Int] = scalaz.concurrent.Task@300555a9
```

你可能会立刻注意到,尽管我们实例化了一个新 Task,但是屏幕上什么也没有打出来。这是比较 Task 和 Future 时的一个重要差异；Future 是立即运算（eagerly evaluated）而 Task 需要你显示的要求才会执行：

```
scala> t.unsafePerformSync
Starting task
res0: Int = 42
```

这里调用 unsafePerformSync 实例方法来执行任务。我们可以看到 println 和返回结果 42。注意 unsafePerformSync 是一个不安全的调用。如果计算抛出异常，会被 unsafePerformSync 会重抛出来（re-thrown）。为了避免这个副作用，更推荐调用 unsafePerformSyncAttemp。unsafePerformSyncAttemp 实例捕捉异常而其返回类型为 Throwable \/ A，让你可以明确地处理失败场景。注意在创建任务 t 时，我们没有提供 ExecutorService。默认情况下，apply 创建的 Task 运行在 DefaultExecutorService 上，这是一个固定线程数的线程池，大小基于机器的可用核数。DefaultExecutorService 类似于 Future 里面全局的 ExecutorContext。它也是 CPU 密集型的，其大小基于机器的可用核数。我们也可以在创建的时候提供一个不同的 ExecutorService:

```
scala> val es = Executors.newFixedThreadPool(4)
es: java.util.concurrent.ExecutorService =
java.util.concurrent.ThreadPoolExecutor@4c50cd8c[Running, pool size = 0,
active threads = 0, queued tasks = 0, completed tasks = 0]

scala> val t = Task {
 println("Starting task on thread " + Thread.currentThread.getName)
 40 + 2
}(es)
t: scalaz.concurrent.Task[Int] = scalaz.concurrent.Task@497db010
```

```
scala> println("Calling run from " + Thread.currentThread.getName)
Calling run from run-main-1

scala> t.unsafePerformSync
Starting task on thread pool-8-thread-2
res2: Int = 42
```

输出显示 Task 在提供的 ExecutorService 执行而不是在主线程上。

谈到 Task 的执行，让我们做一点实验。来创建一个 Task 实例，然后调用两次 unsafePerformSync：

```
scala> val t = Task {
     |   println("Starting task")
     |   40 + 2
     | }
t: scalaz.concurrent.Task[Int] = scalaz.concurrent.Task@300555a9
scala> t.unsafePerformSync
Starting task
res0: Int = 42

scala> t.unsafePerformSync
Starting task
res1: Int = 42
```

我们注意到每次调用 unsafePerformSync 都打印了 Starting task。这意味着每次调用 unsafePerformSync 时都执行了完整计算。这是和 Future 的另一个不同。Future 记得计算结果，而在每次调用 unsafePerformSync 时重新执行计算。换句话说，Task 是引用透明的，因此，比 Future 更接近函数式编程范式。

异步行为

类似 Future，我们能够（而且推荐）用异步的方式使用 Task。Task 实例可以用 unsafePerformAsync 来异步执行。这个方法接受一个（Throwable \/ A）=> Unit 类型的回调，会在计算结束的时候调用。观察下面的代码：

```
def createAndRunTask(): Unit = {
```

```
  val t = Task {
    println("Computing the answer...")
    Thread.sleep(2000)
    40 + 2
  }

  t.unsafePerformAsync {
    case \/-(answer) => println("The answer is " + answer)
    case -\/(ex) => println("Failed to compute the answer: " + ex)
  }

  println("Waiting for the answer")
}
```

我们创建了一个 Task，然后加了 Thread.sleep 来模拟一个昂贵的计算。我们调用 unsafePerformAsync，用一个简单回调来打印结果（或在计算失败时打印异常）。调用 createAndRunTask 观察到如下输出：

```
scala> TaskExample.createAndRunTask()
Waiting for the answer

scala> Computing the answer...
The answer is 42
```

你可以看到最后一句“waiting for the answer”先打印出来。这是因为 unsafePerform Async 立刻返回。可以看到计算里的语句和回调里打印的结果。这个方法大致上等同于 Scala Future 里的 onComplete。

Task 伴生对象提供的另一个有用方法是 async。还记得前面我们是怎么用 Promise 来把一个基于回调的 API 编程返回 Future 实例的 API 的吗？可以用 Task 达成相同的目标；也就是说可以把基于回调的 API 转化成返回 Task 的更 monadic 的 API：

```
object CallbackAPI {
  def doCoolThings[A](a: => A, f: (Throwable \/ A) => Unit): Unit = ???
}

def doCoolThingsToTask[A](a: => A): Task[A] =
 Task.async { f =>
```

```
    CallbackAPI.doCoolThings[A](a, res => f(res))
  }
```

在 REPL 里执行这个方法产生如下输出：

```
> val t = doCoolThingsToTask(40+2)
> t.map(res => res / 2).unsafePerformSync
res2: Int = 21
```

我们的 doCoolThingsToTask 方法用 Task.async 来从 CallbackAPI 里定义的基于回调的 API 里创建一个 Task 实例。Task.async 甚至可以用来把 Scala Future 变成一个 Scalaz Task：

```
def futureToTask[A](future: Future[A])(implicit ec: ExecutionContext):
Task[A] =
 Task.async { f =>
   future.onComplete {
     case Success(res) => f(\/-(res))
     case Failure(ex) => f(-\/(ex))
   }
 }
```

注意我们必须提供一个能够在 Future 的 onComplete 方法里调用的 ExecutionContext。Future 上定义的几乎所有方法都立刻把计算提交给线程池。



也可以把 Task 转化成 Future：

```
def taskToFuture[A](t: Task\[A]): Future[A] = {
  val p = Promise[A]()
  t.unsafePerformAsync {
    case \/-(a) => p.success(a)
    case -\/(ex) => p.failure(ex)
  }
  p.future
}
```

执行模式

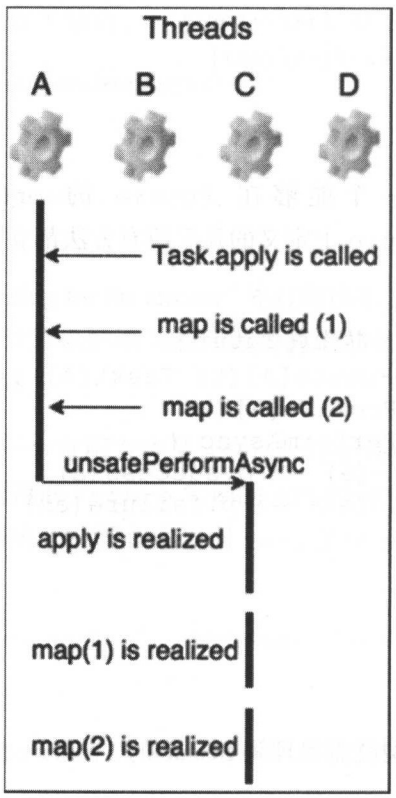
要理解 Task 的执行模型就需要理解 Scalaz 的 Future 执行模型，因为 Task 组合了

Scalaz Future 并添加了错误处理。这一点从 Task 的定义可以看出来：

```
class Task[+A](val get: Future[Throwable \/ A])
```

这个定义里的 Future 不是 Scala 标注库里的版本，而是 Scalaz 提供的一个替代版。Scalaz Future 把转换定义和执行策略解耦，给我们提供了对 Executor 提交的细粒度的控制。Scalaz 通过把自己定义为蹦床运动（trampolining computation）来达成目标。蹦床是指把一个计算描述为一系列使用常量空间运行的离散计算。要理解蹦床的细节，我们推荐阅读 Runar Bjarnason 的论文，*Stackless Scala with Free Monads*，地址 http://blog.higher-order.com/assets/ trampolines.pdf。

Task 在 Scalaz Future 的基础上用 Scalaz \/ disjunction 增加了错误处理。Task 是计算的描述。转换添加在计算的描述上，最终会由线程池执行。要开始运行，Task 必须被显式启动。这个行为很有意思，因为当 Task 最终执行时，我们能够把计算限制在单独一个线程上执行。这提高了线程重用，减少了上下文切换。



在上图里，我们看到几个对 apply 和 map 的调用。这些调用仅仅修改将要执行的任务的描述。只有当我们调用 unsafePerformAsync 时这些计算才实际地在另一个线程上运行。注意所有的转换都在同一个线程里执行。

我们可以做个小基准测试来验证 Future 和 Task 的性能，比较它们基于转换（比如说 map 和 flatMap）的吞吐量和应用的转换的个数。下面展示了部分基准测试代码：

```
@Benchmark
def mapWithFuture(state: TaskFutureState): Int = {
  implicit val ec = state.context
  val init = Future(0)
  val res = (1 until state.operations).foldLeft(init)
    ((f, _) => f.map(_ + 1))
  Await.result(res, Duration("5 minutes"))
}
@Benchmark
def mapWithTask(state: TaskFutureState): Int = {
  val init = Task(0)(state.es)
  val res = (1 until state.operations).foldLeft(init)
    ((t, _) => t.map(_ + 1))
  res.unsafePerformSync
}
```

两个场景都运行类似的计算。我们创建初始的 Future 或 Task 实例，包含 0 值，然后我们应用几个连续的 map 操作来累加 1。另外两个场景执行相同的计算，不过用的是 flatMap。flatMap 的结果显示在下表里：

基准测试	操作数	吞吐量（ops）	错误率（占吞吐量百分比）
flatMapWithFuture	5	41,602.33	± 0.69
flatMapWithTask	5	59,478.50	± 2.14
flatMapWithFuture	10	31,738.80	± 0.52
flatMapWithTask	10	43,811.15	± 0.47
flatMapWithFuture	100	4,390.11	± 1.91
flatMapWithTask	100	13,415.30	± 0.60

Map 操作结果见下表：

基准测试	操作数	吞吐量（ops）	错误率（占吞吐量百分比）
mapWithFuture	5	45,710.02	± 1.30
mapWithTask	5	93,666.73	± 0.57
mapWithFuture	10	44,860.44	± 1.80
mapWithTask	10	91,932.14	± 0.88
mapWithFuture	100	19,974.24	± 0.55
mapWithTask	100	46,288.17	± 0.46

这个基准测试凸显了 Task 由于执行模型不同而带来的性能收益。即使对于较少的转换次数，因为采用了延迟计算方式其吞吐量仍然更好。

用 Task 为交易日模拟建模

掌握 Task 后，我们现在有了足够知识来把单次回测执行改造成并发的了。你可能还记得，从 Dave 那里我们发现 MV 交易公司每天会在交易日结束前清仓。这个发现让我们可以独立地为每个交易日建模。先从模型开始熟悉一下当前实现：

```
case class PnL(value: BigDecimal) extends AnyVal
object PnL {
  def merge(x: PnL, y: PnL): PnL = PnL(x.value + y.value)
  val zero: PnL = PnL(0)
}
case class BacktestPerformanceSummary(pnl: PnL)
case class DecisionDelayMillis(value: Long) extends AnyVal
```

损益（profit-and-loss，简称 PnL）是每个交易日模拟的输出。PnL 提供了方便方法来加总两个 PnL 实例，用来对多个交易日模拟输出的 PnL 求和。一旦所有交易日都模拟完成，就创建一个 BacktestPerformanceSummary 来捕捉整个模拟的损益。对于回测例子，我们将用 Thread.sleep 来模拟昂贵的计算以免实际设计真实的交易策略。sleep 时间的长短用 DecisionDelayMills 来参数化。

下面显示一个简化版的回测，演示怎么用 DecisionDelayMillis 来模拟一个交易日：

```
def originalBacktest(
  testDays: List[MonthDay],
  decisionDelay: DecisionDelayMillis): BacktestPerformanceSummary =
```

```
    {
    val pnls = for {
      d <- testDays
      _ = Thread.sleep(decisionDelay.value)
    } yield PnL(10)
    BacktestPerformanceSummary(pnls.reduceOption(PnL.merge).getOrElse(
      PnL.zero))
  }
```

原本的回测用同步的方式显示一系列日期的模拟过程。为了可重现性，我们用一个常量的损益（10$）替换了动态值。这个回测忽略了时钟和交易策略以便关注于核心问题：怎么用 Task 来给回测加入并发？

从我们的例子可以看出，Task 通过向 ExecutorService 提交多个 Task 和用 unsafePerformAsync 执行 Task 以避免阻塞式等待结果从而达成了并发。作为第一步，先用 Task 实现第一版回测但是不引入并发：

```
  def backtestWithoutConcurrency(
    testDays: List[MonthDay],
    decisionDelay: DecisionDelayMillis): Task[BacktestPerformanceSummary] =
    {
    val ts = for (d <- testDays) yield Task.delay {
      Thread.sleep(decisionDelay.value)
      PnL(10)
    }
    Task.gatherUnordered(ts).map(pnls => BacktestPerformanceSummary(
      pnls.reduceOption(PnL.merge).getOrElse(PnL.zero)))
  }
```

这个实现把返回类型变成了 Task[BacktestPerformanceSummary]。因为没有运行 Task，这个方法里面保持了引用透明性。每个交易日用 Task.delay 来模拟。delay 是 Task.now 的 Lazy 变种，推迟运算提供的值。我们看一下方法签名以确认这一点：

```
def delay[A](a: => A): Task[A]
```

如果我们用 Task.now 而不是 Task.delay，休眠（代表模拟）会在运行 Task 前就生效。这里还能看到我们使用了另一个新功能，Task.gatherUnordered. 当你想做如

下转换时 gatherUnordered 是有用的：

```
List[Task[A]] => Task[List[A]]
```

虽然这里用了 List，这个关系对于任何 Seq 都成立。gatherUnordered 提供了一种方法来把任务集合变成一个单一的包装了里面的类型集合的任务。让我们看看下面这个方法签名来加深理解：

```
def gatherUnordered[A](tasks: Seq[Task[A]], exceptionCancels: Boolean =
false): Task[List[A]]
```

这个签名紧密贴合我们前面定义的函数，多了一个可选的 Boolean 参数。当 exceptionCancels 设置为 true，任何等待中的 Task 也不会执行。gatherUnordered 让我们能够把每个交易日的损益结果合并返回单独一个包装了 BacktestPerformanceSummary 的 Task。Scala Future 伴生对象提供了一个类似的方法，叫作 sequence，对 Future 序列执行相同的操作。

这是一个可工作的回测实现，但是没有给历史交易日模拟增加并发。在下一个迭代，我们将利用 Task API 的另一个功能，Task.fork。先来看看我们怎么使用它，然后解释它的原理：

```
def backtestWithAllForked(
  testDays: List[MonthDay],
  decisionDelay: DecisionDelayMillis): Task[BacktestPerformanceSummary] =
  {
  val ts = for (d <- testDays) yield Task.fork {
    Thread.sleep(decisionDelay.value)
    Task.now(PnL(10))
  }
  Task.gatherUnordered(ts).map(pnls => BacktestPerformanceSummary(
    pnls.reduceOption(PnL.merge).getOrElse(PnL.zero)))
}
```

这个实现和以前一样搜集交易日 PnL，但是这次用了 Task.fork 和 Task.now 的组合来模拟交易日。让我们看看 Task.fork 的签名来理解运行时行为发生了什么变化：

```
def fork[A](a: => Task[A])(implicit pool: ExecutorService =
Strategy.DefaultExecutorService): Task[A]
```

fork接受一个Task作为by-name参数和一个隐式的ExecutorService，默认适用于CPU密集型运算。签名显示fork把传入的Task提交给pool以便把计算分发到另一个线程上。fork是一种显式的方式来控制Task的并发。概念上说，fork类似于任何包含把任务提交到Executor的Future转换（比如map）。因为fork懒惰地运算其参数，可以用Task.now来把交易日损益提升进Task。在这种实现方式下，代表每个交易日的Task被提交给executor。我们假设正在回测30个交易日而计算机有两个核，那么这个实现让每个核模拟15个交易日而不是让一个交易日模拟30个。

如我们在前面的基准测试中所见，把大量的小计算提交给executor是昂贵的。如果能显式地用fork来控制Task的并发度，我们就能优化executor提交来提升性能。在第3版里，我们利用已经知道交易日的数量这个优势来控制executor提交的频率。实现如下：

```
def backtestWithBatchedForking(
  testDays: List[MonthDay],
  decisionDelay: DecisionDelayMillis): Task[BacktestPerformanceSummary] =
  {
  val ts = for (d <- testDays) yield Task.delay {
    Thread.sleep(decisionDelay.value)
    PnL(10)
  }
  Task.gatherUnordered(ts.sliding(30, 30).toList.map(xs =>
    Task.fork(Task.gatherUnordered(xs)))).map(pnls =>
    BacktestPerformanceSummary(
      pnls.flatten.reduceOption(PnL.merge).getOrElse(PnL.zero)))
}
```

这个实现回到了用Task.delay无并发地模拟每个交易日的方式。跟之前的实现不同的是，交易日模拟列表Tasks被用sliding分割成30个块（chunks of 30）。每份30个块用一次 Task.fork 调用包装来并发执行。这种方式让我们可以在并发的收益和executor提交的开销之间取得平衡。

在这3种实现里，哪种性能最好？没有简单的答案，因为要看交易日模拟的量和每个交易日模拟的计算成本多大。为了更好地理解这个取舍，我们写了一个微基准来测试这3种实现。我们展示了运行基准所需的状态：

```
@State(Scope.Benchmark)
class BenchmarkState {
  @Param(Array("1", "10"))
  var decisionDelayMillis: Long = 0
  @Param(Array("1", "12", "24" ))

  var backtestIntervalMonths: Int = 0
  var decisionDelay: DecisionDelayMillis = DecisionDelayMillis(-1)

  var backtestDays: List[MonthDay] = Nil
  @Setup
  def setup(): Unit = {
    decisionDelay = DecisionDelayMillis(decisionDelayMillis)
    backtestDays = daysWithin(trailingMonths(backtestIntervalMonths))
  }
}
```

这个基准让我们扫测不同间隔的回测和时延决定的组合。用 daysWithin 方法（代码里省略了），一个代表月份个数的计数器被转化成一个交易日模拟列表。我们只展示一个基准测试的实现，因为其他两个是一样的：

```
@Benchmark
def withBatchedForking(state: BenchmarkState): BacktestPerformanceSummary =
  Backtest.backtestWithBatchedForking(state.backtestDays,
  state.decisionDelay)
    .unsafePerformSync
```

为了准确地给完成一个 Task 计算的时间计时，我们用阻塞的 unsafePerformSync 方法来开始计算。可是少数可以接受不带超时做阻塞调用的场景。对于这个受控的例子，我们确信所有调用都会返回。对于这个测试，我们扫测不同月数，把决策时延固定在 1 毫秒。在一台四核机器上运行这个基准测试得到如下结果：

基准测试	月	决策时延（毫秒）	吞吐量（ops）	错误率（占吞吐量百分比）
withoutConcurrency	1	1	25.96	±0.46
withAllForked	1	1	104.89	±0.36
withBatchedForking	1	1	27.71	±0.70

续表

基准测试	月	决策时延（毫秒）	吞吐量（ops）	错误率（占吞吐量百分比）
withoutConcurrency	12	1	1.96	± 0.41
withAllForked	12	1	7.25	± 0.22
withBatchedForking	12	1	8.60	± 0.49
withoutConcurrency	24	1	0.76	± 2.09
withAllForked	24	1	1.98	± 1.46
WithBatchedForking	24	1	4.32	± 0.88

测试结果使批量化（batching）的开销和收益之间的取舍明显可见。在计算时延很短只有 1 毫秒的场景下，随着月数的增加，批量化是显然的赢家。考虑回测 24 个月 1 毫秒决策时延的场景，假设每个月 30 天，就有 720 个交易日要模拟。通过分割成 30 个批量，总共有 24 次 fork 调用而不是 720 次。把 Task 分批和搜集每个批次结果的开销比 executor 提交的开销小一个数量级因而被遮盖了。在这个场景下，我们通过显式控制提交使吞吐量提高了一倍。

随着约束的减少，创建 Task 批次的开销开始占大头。在 12 个月的回测里，总共有 360 个交易日，产生 12 个批次。在这个场景下，批量化比提交所有的任务有 20%的吞吐量提升。交易日数量从 24 个月减半是性能优势下降超过一半。在更差的场景下，如果只模拟一个月，批量化策略就不能充分利用机器的核数了。在这种场景下，只创建一个批次，使 CPU 利用率不足。

总结回测

如我们所见，有好几个因素在这里起作用。权衡计算成本、可用核数、预期 Task executor 提交次数和批量化开销是一个挑战。对了扩展我们的工作，我们可以调研一种更动态的批量化策略来用更小的回测间隔更好地利用 CPU 资源。通过这个基准测试，我们对 Task 提供的额外工具和显式控制 executor 提交对吞吐量的影响有了实际体会。

通过回测所得到的洞察可以应用在更大规模的软件系统上。我们之前专注于分析决策时延为很多的 1 毫秒场景下的基准测试结果。随着执行每个 Task 的成本增加（比如 10 毫秒决策时延），从批量化上获得的性能提升下降。这是因为 executor 提交的成本被执行计算的成本遮盖了。虽然 1 毫秒看上去很短，在 1 毫秒里能完成的计算量比你以为的要大得多。想想 1000 操作/秒等于 1 操作/毫秒。回忆前面章节做过的基准测试以及你自己的实际日常

工作场景，你可以找到无数例子的吞吐量高于 1 操作/毫秒。通过这个思想实验可以发现大量场景符合短时间计算（比如 1 毫秒）的特征，也就意味着有大量的机会通过合理使用 fork 来优化并发。



回测是批量化的一个初级的例子，因为工作量，也就是待模拟的天数，在执行前是已知的。在流式处理环境下，工作量是未知的。比如说，考虑到指令簿随时接受事件。你要怎么在流式环境下实现批量化？

我们希望回测提供了一个有代表性的例子来让你感受一下 `Task`。`Task` 还提供了别的工具我们没有探讨。请你自己阅读 Scalaz 库的 `Task` 文档。在两位 Scalaz 贡献者，Runar Bjarnason 和 Paul Chiusano 写的 *Functional programming in scala* 一书中，有很棒的一章描述一个简化版的 Scalaz `Task` 的实现。这是理解 `Task` API 设计的极佳资料。

小结

在本章里，我们学习了如何用 Scala 标准库里的 `Future` 和 `Promise` 驾驭异步编程的威力。我们通过引入并发来提高了 MV 交易公司的回测性能，并学习了如何用 `Promise` 来扩展 `Future`。在过程中，我们调研了 `Future` 的缺陷和应对这些缺陷的技巧。还探索了 Future 的替代品——Scalaz `Task`，它提供了类似的性能收益同时保持了引用透明性。利用你在本章里学到的知识，你能够用 Scala 充分发挥多核硬件的优势来扩展你的系统，交付更好的吞吐量。在最后一章——第 7 章，我们将探讨一组高级函数式编程技巧和概念来丰富你的函数式编程工具箱。

7

高性能架构

我们已经花了很多时间来探索如何用 Scala 和各种技巧来编写高性能代码。在这最后一章，我们来讨论没有标准答案的主题。这些主题的使用场景大都不局限于 Scala 和 JVM。我们将深入多种工具和实践来提升应用的设计和架构。在本章里，将讨论如下主题：

- 无冲突复制数据类型（Confilct-free replicated data types——CRDTs）
- 使用队列对吞吐量和时延的影响
- Free monad

分布式自动化交易员（Distributed automated traders）

由于我们的努力，MVT 的业务蒸蒸日上。销售部门没命地签订单，电话铃从早响到晚。指令簿现在能够处理更多指令，随着交易量的上升，MVT 的另一个产品开始出现性能问题：自动交易系统。自动交易员从指令簿接受指令，然后实时应用各种交易策略来代表客户自动下指令。指令簿现在能够处理的指令数比原来高了一个数量级，而自动交易系统的性能跟不上，因此无法有效地应用交易策略。由于算法做出的错误判断和高执行时延，最近有几个大客户损失了大笔资金。工程师团队必须解决这个性能问题。Alice，你的技术领导，指定你找出解决方案，防止公司失去刚刚获得的客户。

在上一章里，我们学习并应用了并发技术，学到了如何设计代码来发挥多核硬件的能力。自动化交易员已经为并发做了优化，能够利用机器的全部 CPU 资源。事实是，一台机器即使是多核的也有其能力极限。为了扩展系统以便跟上来自指令簿的流量增长，我们必须开始实现分布式系统。

分布式架构概述

分布式计算是个大课题，我们不可能在一章里讲完它。本章对分布式计算做一个简要、不完备的描述。我们会尝试让你对分布式计算范式有个整体认识，指出分布式系统的一些主要收益和挑战。

分布式计算背后的思想是把系统设计成多个组件，每个组件在不同的机器上运行并互相通信（比如通过网络）来执行任务或提供服务。分布式系统可以包含有不同特点的组件，每个组件提供某种特定服务并参与整个任务的实现。比如说，可以部署一个 web 服务器来接受 HTTP 请求。为了处理请求，web 服务器可能会通过网络查询身份验证服务来鉴权，查询数据库服务器来存取数据以便完成请求。web 服务器、身份验证服务和数据库共同构成了一个分布式系统。

分布式系统的同个组件可能有多个实例。这些实例构成一个节点集群，用来分担工作负载。这种拓扑使系统可以横向扩展（scale out），通过增加实例来支持更高的负载。比如说，如果一台 web 服务器可以处理 20000 请求/秒，那跑 3 个服务器的集群可能就能处理 60 000 请求/秒（假如你的架构允许你的应用线性扩展）。分布式集群还能帮助达成高可用的目标。如果一个节点挂了，在挂掉的节点重启或恢复的时候，其他的节点还在运行并处理请求。因为没有单点故障（single-point of failure），所以服务永不中断。

在带来这么多好处的同时，分布式系统也有其缺点和挑战。组件间的通信容易失败，网络也可能会中断。应用需要实现重试和错误处理机制，还要能处理消息丢失的情况。另一个挑战是管理共享状态。比如说，如果所有的节点用单独一个数据库服务器来存取信息，那么数据库必须实现某种形式的加锁机制来防止并发修改。另一方面，当集群节点数量增长到足够大时，数据库会无法高效伺服全部请求从而变成一个瓶颈。

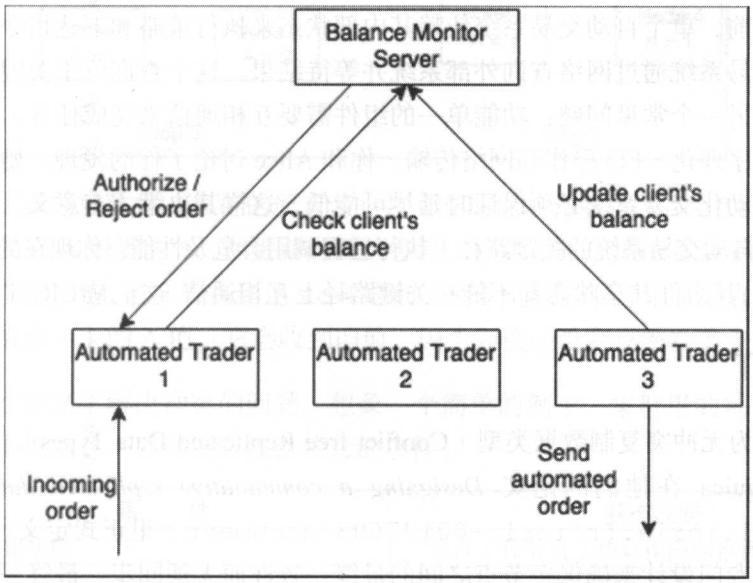
你已经简要了解了什么是分布式系统，现在我们回到 MV 交易公司。工程师团队已经决定把自动化交易系统改造成分布式应用来扩展系统。你受命设计这个系统。该到白板边来规划一番了。

第一次尝试分布式自动化交易系统

你先尝试简单的策略——部署自动化交易系统的多个实例来构成节点集群。这些节点可以分担工作，各自处理完整请求。在集群前加个负载均衡来平均分配任务。这个新架构

有助于横向扩展交易系统。但是，你碰到了分布式系统的常见问题：节点们必须共享一个公共状态来运作。为了理解这个问题，我们来探索自动化交易系统的一个特性。在能够使用 MV 交易公司的自动交易系统前，客户需要在 MV 交易公司开个账户并预付足够的钱来支持他们的交易。MV 交易公司以此作为安全网，以防客户赖账。为了确保自动化策略不会超支，自动化交易系统保持跟踪每个客户的当前余额并在替客户下达自动化指令前检查余额。

你计划部署自动交易系统的多个实例。每个实例接受指令簿处理的指令中的一部分，运行某个策略，然后代表客户下达匹配的指令。现在系统包含几个并行运行的实例，多个实例都可能替同一个用户下指令。为了执行余额校验，每个实例都需要关注所有客户的当前余额。客户余额成为必须在集群里同步的共享状态。为了解决这个问题，你想到可以部署一个单独的余额监控服务器来保存每个客户的余额。当自动交易系统集群的节点收到交易指令时，节点会请求余额监控服务器来检查客户账户是否有足够的资金来下达自动化交易指令。类似地，当交易执行时，节点通知余额监控服务器来更新客户余额。



上图描述了在你的架构下各组件之间的交互。Automated Trader 1 接受一个交易指令，然后查询余额监控服务器来检查客户是否有足够的资金来执行交易。余额监控服务器要么授权要么拒绝指令。在同一时间，Automated Trader 3 发出之前被余额监控服务器审核通过的指令同时更新客户余额。

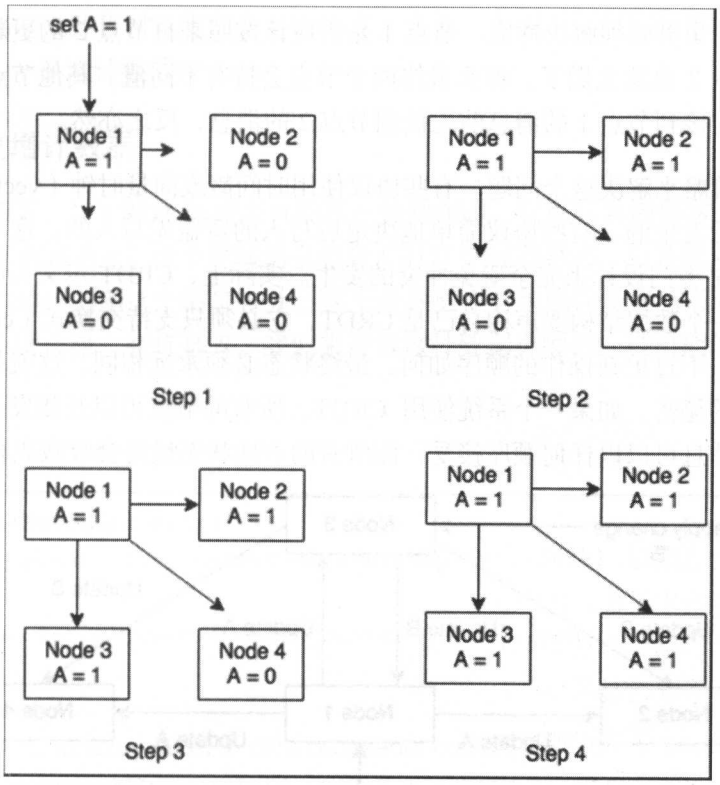


你很可能已经注意到这个设计里的一个瑕疵。自动交易系统的两个不同实例可能同时发起对同一个客户的余额校验（译者注：此时客户余额够一笔交易使用，但由于校验时并不更新余额，所以两个校验都通过了），各自从 Balance Monitor Sever 获得授权，然后两个都下达交易指令，结果超出客户账户限制。这类似于在单机上写并发应用时会碰到的竞争条件问题（race condition）。在实践中，这个风险很小，类似 MV 交易公司这样的公司一般会接受这个风险。给客户设置的限额通常设置为小于账户的实际余额以便应对这种风险。把系统设计成能处理这种场景会增加系统的时延，因为我们将不得不引入更严格的节点间同步机制。这个例子很好地说明了技术和业务专家精诚合作来优化方案的重要性。

在设计讨论结束时，你停下来边踱步边理清思路，顺便喝瓶苏打水。当你回到白板时，一个残酷的现实击中了你。你意识到这些长方形之间的连线在现实中都代表着在网络上穿行的消息。当前，单个自动交易系统依赖其内部状态来执行策略和下达指令，而你的新设计需要自动交易系统通过网络查询外部系统并等待结果。这个查询位于关键路径上。这是分布式系统的另一个常见问题：功能单一的组件需要互相通信来完成任务。这种通信是有代价的，包括序列化、I/O 操作和网络传输。你和 Alice 讨论了你的发现，她也确认这确实是个问题。自动化交易系统必须保证时延尽可能低，这样其决策才有意义。在简短的讨论后，你同意在自动交易系统的关键路径上执行远程调用会危及性能。你现在的任务是实现一个分布式系统，其组件共享状态却不能在关键路径上互相通信。这正是 CRDT 的用武之地。

引入 CRDT

CRDT 意为无冲突复制数据类型（Conflict-free Replicated Data Types）。Marc Shapiro 和 Nuno Preguica 在他们的论文 *Designing a commutative replicated data type*（参考 `https://hal.inria.fr/inria-00177693/document`）里正式定义。CRDT 是一种数据结构，专门设计来确保多节点之间的最终一致性而无须同步。最终一致性是分布式领域的众所周知的概念，跟 CRDT 不冲突。这个模型确保最终，如果一份数据不再被修改，集群的所有节点持有的数据最终会具有相同的值。节点间互相发更新通知来保持它们的状态同步。和强一致性的区别在于，在给定时间，一些节点可能会看到略微过时的数据，直到他们接到更新通知。



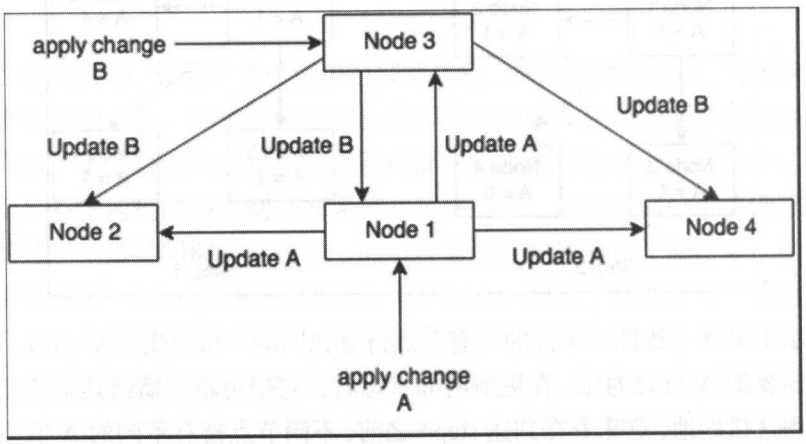
上图展示了最终一致性。集群的所有节点持有相同的一份数据（A = 0）。Node 1 接到一个更新通知要把 A 设置为 1。在更新内部状态后，它把更新广播给其余节点。更新消息在不同时间到达目的地，意味着在到达 step4 之前，不同节点持有不同的 A 值。如果在 step3 时，客户查询节点 4 上的 A 值，它会收到旧值，因为新值还没有反映在节点 4 上。

最终一致性带来了解决冲突的问题。想象一个简单的例子，集群里的每个节点共享一个整数数组。下表描述一系列更新数组状态的事件：

时间点	事　　件	状态改变
T0	初始化集群	节点 1 和节点 2 持有相同的整数数组：[1,2,3]
T1	节点 1 接到请求把下标 1 的值从 2 改成 4	节点 1 更新其内部状态为[1,4,3]，然后发送更新消息给节点 2
T2	接到请求把下标 1 的值从 2 更新为 5	节点 2 更新其内部状态为[1,5,3]，然后发送更新消息给节点 1
T3	节点 1 接到来自节点 2 的更新消息	节点 1 必须决定是忽略还是采纳节点 2 的更新消息

现在我们的集群必须解决冲突。节点 1 是否应该按照来自节点 2 的更新通知来更新其状态？如果节点 2 也这么做了，那么最终两个节点会持有不同值。其他节点会怎样呢？其中有些可能会在接到节点 1 的消息前先接到节点 2 的消息，反之亦然。

存在多种策略来解决这个问题。有些协议使用时间戳或向量时钟（vector clock）来判断哪个更新是后发生的。有些协议简单地决定后写入的覆盖先写入的。这不是一个简单的问题，而 CRDT 专门设计来完全避免冲突的发生。实际上，CRDT 定义为使冲突在数学上不可能。如果一个数据结构要声称自己是 CRDT，它必须只支持交换式（commutative）更新。也就是说，不过更新操作的顺序如何，最终状态必须永远相同。这就是没有合并冲突的最终一致性的秘密。如果一个系统使用 CRDT，所有的节点可以互相发送更新消息而无须严格同步。消息可以以任何顺序接受，而所有的本地状态最终会收敛成相同值。



在上图里，我们看到 Node3 和 Node1 接到两个不同的修改。他们把这个更新信息发送给所有其他节点。注意我们没有顾虑其他节点接到更新的顺序。由于更新是交换式的，它们的顺序对每个节点计算出来的最终结果没有影响。一旦所有节点都接到所有的更新广播，就能确保所有节点持有相同数据。

存在两种类型的 CRDT：

- 基于操作的
- 基于状态的

这两者是等价的，对于每个基于操作的 CRDT 总是能够定义出相应的基于状态的 CRDT，反之亦然。但是它们的实现不同，并且对于错误恢复和性能提供了不同的保证。

我们两种类型都定义并且讨论它们各自的特点。作为一个例子，我们实现两种版本的最简单的 CRDT：一个只增计数器。

基于状态的只增计数器

在这个模型下，当 CRDT 接到来自的客户的操作时，它更新其内部状态并给集群里的所有 CRDT 发送更新消息。这个更新消息包含 CRDT 的完整状态。当其他 CRDT 接到这个消息后，把它们的状态和这个消息合并。这个合并操作必须确保最终状态永远相同。它必须符合交换律、结合律，并且幂等。我们来看看这种数据类型的一种可能的实现。

```
case class CounterUpdate(i: Int)
case class GCounterState(uid: Int, counter: Int)

class StateBasedGCounter(
 uid: Int, count: Int, otherCounters: Map[Int, Int]) {

 def value: Int = count + otherCounters.values.sum

 def update(
   change: CounterUpdate): (StateBasedGCounter, GCounterState) =
   (new StateBasedGCounter(uid, count + change.i, otherCounters),
    GCounterState(uid, count))

 def merge(other: GCounterState): StateBasedGCounter = {
   val newValue = other.counter max otherCounters.getOrElse(other.uid,0)
   new StateBasedGCounter(uid, count, otherCounters.+(other.uid ->
newValue) )
 }
}
```

客户可以用 update 方法来增加 counter 的值。这会返回一个新的基于状态的计数器，包含更新后的值，并且生成一个 CounterState 对象，可以发送给集群里的所有其他 CRDT（译者注：这里似乎有 bug，返回的 GCounterState 里的 count 值是更新前的 count 值）。merge 方法用来处理这些 GCounterState 消息，把来自其他计数器的新状态和本地状态合并。每个计数器在集群里有一个唯一 ID。内部状态由本地状态（也就是 count 值）和集群里所有其他计数器的状态组合而成。我们用一个 map 来保存这些计数

器，每当我们接到不同计数器来的状态信息时，在 merge 方法里更新这个 map。合并是个简单操作。我们比较传入值和 map 里已有的值，取较大的一个。这是为了确保在接到两个顺序错误的更新消息时，我们不会用延迟的旧更新信息覆盖最新的状态(也就是较大的数字)。

基于操作的只增计数器

基于操作的 CRDT 类似于基于状态的 CRDT，区别在于更新消息只包含刚刚执行的操作的描述。这种 CRDT 不在更新消息里发送其完整状态，而仅仅包含它们刚才执行来更新其内部状态的操作的拷贝。这确保集群里的所有其他 CRDT 执行相同的操作，从而维持其状态一致。集群里的每个节点可以以不同顺序接收更新消息。为了确保最终状态一致，更新必须符合交换律。这种数据结构的例子如下：

```
class OperationBasedCounter(count: Int) {

 def value: Int = count

 def update(change: CounterUpdate): (OperationBasedCounter, CounterUpdate)
 =

   new OperationBasedCounter(count + change.i) -> change

 def merge(operation: CounterUpdate): OperationBasedCounter =
   update(operation)._1

}
```

(译者注：在 scala 里，(x,y) 和 x -> y 是一样的，所以 update 的返回结果是符合类型签名的)。

这个实现比基于状态的例子短。update 方法还是返回更新后的计数器实例和 CounterUpdate 对象。对于基于操作的计数器，只要把操作广播出去就可以了。其他计数器实例接到更新时使用 merge 方法来对它们的内部状态应用相同的操作。注意 update 和 merge 是等价的，merge 甚至是基于 update 实现的。在这种模型下，不需要全局唯一的计数器 ID。

基于操作的 CRDT 发送的消息体一般较小，因为它们只发送每个离散的操作而不是完

整的内部状态。在我们的例子里，基于状态的更新包含两个整数，而基于操作的更新只包含一个。较小的消息体有助于减小带宽占用和提高系统吞吐率，但是它们对通信失败敏感。如果一个更新消息在传输时丢失，没有到达某个节点，这个节点就会和集群里的其他节点失去同步而且无法恢复。如果你决定采用基于操作的 CRDT，你必须能够信任你的通信协议，确信所有更新消息都能到达目的地并被正确处理。基于状态的 CRDT 没有这个问题，因为它们总是在更新消息里发送完整状态，如果一个消息丢失，没有到达目标节点，该节点只会暂时不同步，等到接到下一个更新消息时就会恢复同步。还可以通过定期广播（即使没有更新事件）节点的状态使这种模型更加可靠。通过这种方式可以强迫所有节点周期性地发送它们的当前状态，从而确保整个集群总是能够最终一致。

CRDT 和自动化交易系统

基于系统的要求，看上去 CRDT 是一种合适的实现方式。每个节点可以在内存里类似计数器一样保存客户的余额状态，在下达指令时更新并且把更新消息广播给系统的其他节点。广播可以在关键路径外进行，而且我们不需要担心冲突处理问题，因为这是 CRDT 设计来解决的问题。最终所有节点会在内存里持有每个客户余额的相同的值，从而可以在本地检查交易授权。余额监控服务器可以完全移除。

为了把余额实现成 CRDT，我们需要比前面那个计数器更精巧的设计。余额不能用只增计数器来代表，因为订单偶尔会被取消，此时系统必须回充客户账户余额。余额计数器必须能同时处理增加和减少操作。幸运的是，这样的计数器是存在的。我们来看看基于状态的计数器的简单实现：

```
case class PNCounterState(incState: GCounterState, decState: GCounterState)

class StateBasedPNCounter private(
 incCounter: StateBasedGCounter,
 decCounter: StateBasedGCounter) {

 def value = incCounter.value - decCounter.value

 def update(change: CounterUpdate): (StateBasedPNCounter, PNCounterState) =
{
   val (newIncCounter, newDecCounter, stateUpdate) =
     change match {
```

```
        case CounterUpdate(c) if c >= 0 =>
          val (iC, iState) = incCounter.update(change)
          val dState = GCounterState(decCounter.uid, decCounter.value)
          (iC, decCounter, PNCounterState(iState, dState))
        case CounterUpdate(c) if c < 0 =>
          val (dC, dState) = decCounter.update(change)
          val iState = GCounterState(incCounter.uid, incCounter.value)
          (incCounter, dC, PNCounterState(iState, dState))
      }

    (new StateBasedPNCounter(newIncCounter, newDecCounter), stateUpdate)
  }
  def merge(other: PNCounterState): StateBasedPNCounter =
    new StateBasedPNCounter(
      incCounter.merge(other.incState),
      decCounter.merge(other.decState)
    )
}
```

PN counter 利用我们前面实现的只增计数器来实现减数功能。为了把计数器表示为基于状态的 CRDT，我们需要同时对增加和减少操作保持跟踪。为了确保在接到来自其他节点的顺序错误的消息时不丢失信息，这样做是必需的。



回忆一下，只增计数器通过认定较大的计数器值是最新的来解决冲突。这在 PN counter 场景下是不适用的。

这个实现展示了 CRDT 的另一个有趣的性质：简单而基础的数据结构可以组合来构造复杂而特性丰富的 CRDT。是否需要再演示一下如何实现基于操作的 CRDT？实际上，我们相信你应该已经发现前面的只增计数器可以直接支持减值操作。基于操作的计数器可以应用正数或复数的差异值。

当余额不足时

你已经完成了概念验证原型（proof-of-concept）并打电话给 Alice 获取反馈。她花了几分钟研究你的新设计和代码，“看上去不错。别忘了账户黑名单也要同步。”她在说啥？“检

查账户余额只是自动交易通过还是拒绝的检验条件之一。客户的其他属性也要纳入考量。当前版本的自动交易系统在后台运行一个信任算法，给每个客户计算一个信任分。如果分数低于某个阈值，这个账户就被加入黑名单，直到交易日结束，在这期间所有自动交易指令被拒绝。我喜欢你的设计，不过你要把黑名单也加入这个新系统里。”面对这个新挑战，你认为最佳方案是把黑名单也实现为 CRDT，只要它能符合你当前的设计。

一个新的 CRDT——只增集（grow-only set）

一个新的 CRDT 被设计出来应对这个新用例。只增集数据类型实现了 set，只支持添加新元素，不包含重复元素。我们可以把黑名单设计成一个只增集。每个节点可以运行自己的信任算法来判断一个客户是否应该被加入黑名单并在该交易日拒绝其自动交易指令。交易日结束时，系统可以清空黑名单。下面代码展示了一个基于状态的只增集的一种实现方式：

```
case class AddElement[A](a: A)
case class GSetState[A](set: Set[A])
class StateBasedGSet[A](val value: Set[A]) {
 def contains(a: A): Boolean = value.contains(a)
 def update(a: AddElement[A]): (StateBasedGSet[A], GSetState[A]) = {
   val newSet = new StateBasedGSet(value + a.a)
   (newSet, GSetState(newSet.value))
 }
 def merge(other: GSetState[A]): StateBasedGSet[A] = {
   new StateBasedGSet(value ++ other.set)
 }
}
```

我们的实现支持通过调用 update 方法来添加新元素。它返回一个包含更新后状态的新的 StateBasedGSet 实例和一个 GSetState 实例用来广播给其他节点。这个更新信息包含计数器的完整状态，也就是内部的只增集。基于操作的版本比较简单，留给读者作为家庭作业（在本书的代码库里提供了一个实现）。和前面介绍的增量-减量-计数器类似，也可以设计一个 set 同时支持添加和移除元素。给个警告：添加和移除元素并不符合交换律，操作之间存在优先级。 在实践中，可以构造一个 2P 集来支持添加和移除操作，但是一个元素一旦移除就不能再次被加入。移除操作优先级高，以便确保操作符合交换律从而避免冲突处理。一种可能的实现方案是组合两个只增集，一个用来添加元素，另一个用来移除。再一次地，我们可以看到将多个简单 CRDT 组合起来创造更强力的数据类型的威力。

免费交易策略性能提升

你瞪着敏捷燃尽图（burn down chart），发现在 sprint 结束前一天就已经完成了所有的故事点。能够提前提交本周的特性让你感到兴奋，但是又有点担心是不是又得和 scrum master 聊聊故事点预估的事了。你决定不再浪费精力担心预估的事，而是把注意力回到 Dave 提出的一个问题上来。在最近一次聚餐的时候，Dave 提到由于信息过时的原因导致公司的交易策略亏钱的事情。几毫秒的延迟可能意味着极高收益和极大损失的差异。他的话刺激你的兴趣想寻找是否还有能够提高 MV 交易公司的交易策略性能的办法。

MV 交易公司的交易策略是指令簿的下游消费者。交易策略监听最佳出价与入价（best bid and offer——BBO）来判断提交买卖指令的时机。在聚餐上，Dave 解释说跟踪 BBO 在历史上证明能给交易策略最重要的信号。最佳出价是指价格最高的出价，最佳入价是指最低价格入价。当 BBO 的任何一方发生变化——由于取消、执行或新的限制指令——那么一个 BBO 更新时间会被传递给下游的交易策略系统。代表这个事件的模型是 BboUpdated，如下：

```
case class Bid(value: BigDecimal) extends AnyVal
case class Offer(value: BigDecimal) extends AnyVal
case class Ticker(value: String) extends AnyVal
case class BboUpdated(ticker: Ticker, bid: Bid, offer: Offer)
```

MV 交易公司把每个交易策略部署在独立的 JVM 下，以确保一个策略出错不会影响到其他的策略。在部署状态下，每个交易策略维护对一组交易的订阅。

在花费了相当长时间在指令簿上之后，你希望能找到机会应用函数式编程知识来提高性能。在和 Dave 吃饭时，你发现“更好的性能”对于交易策略系统的含义和其他系统有着微妙的区别。你问 Dave，“如果要你在时延和吞吐量之间选一个，你会选哪个？”Dave 敏感地答道，“为什么我非得选一个？我两个都要！”最后，他说道，“时延！几乎每次当交易系统基于过时的 BBO 更新事件来做决策时都导致我们亏钱。事实上，如果可能，我宁可扔掉过时的 BBO 更新事件。我们做高频交易，所以就一定会立刻看到下一个 BBO 更新”。当你在查看代码库时，你在想是否能应用 Dave 的思想来提高交易策略的性能。

为交易策略做基准测试

回忆起在处理指令簿时的经验教训，你决定首先要做基准测试。你选择了生厂上用的

交易策略，然后适配你之前为指令簿写的一个基准测试，FinalLatencyBenchmark，来把 BboUpdated 事件发送给交易策略系统。原本的基准聚焦于显示第 99 百分位以上的时延。既然你已经知道这次性能调研的最主要指标是时延，你修改了基准测试，同时显示中位数和第 75 百分位数的时延。这可以让你对交易策略系统的性能有更整体的理解。

通过查看生产监控系统数据，你找到了一组想做基准的时间序列交易数据。数据显示哪一天交易量较小，只有大约每秒 4 000 个 BBO 更新事件。你挖掘历史监控数据查找最近几周交易量最大的日子。市场又处于波动中，所以最近的高交易量日很可能适合用来做高吞吐量基准测试。 在大约两周前，有个交易日的峰值达到了每秒 12000BBO 更新事件。你就会先从低端，也就是 4000 事件/秒开始，递增到 12000 事件/秒，看看性能会有什么变化。

测试方法是度量不同吞吐量下相同事件数的时延，同时确保仔细测试每个吞吐量级别。为了达到这个目标，你把最高吞吐量，12 000 事件/秒乘以 30 轮，总共 360 000 个事件。在模拟生产环境的测试环境下运行测试的结果显示在下表中。事件/每秒缩写为 EPS：

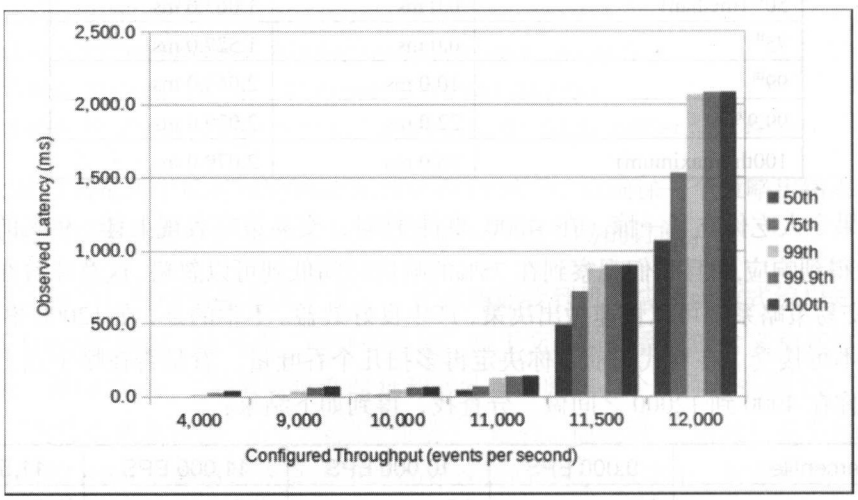
Percentile	4,000 EPS	12,000 EPS
50th (median)	0.0 ms	1,063.0 ms
75th	0.0 ms	1,527.0 ms
99th	10.0 ms	2,063.0 ms
99.9th	22.0 ms	2,079.0 ms
100th (maximum)	36.0 ms	2,079.0 ms

这结果令人吃你吃了一惊。在 4000 事件/秒时，交易策略表现上佳。99%的事件都在 10 毫秒内得到响应,而且我们观察到在 75%的响应时间低到可以忽略。这意味着在低交易量的日子，交易策略系统可以快速做出决策，产生良好效益。不幸的是，在 12000 事件/秒时，性能变得不可接受。在看代码前，你决定再多扫几个吞吐量，看是否在哪个点上性能发生了突变。你在 4000 到 12000 之间做二分查找，得到如下结果：

Percentile	9,000 EPS	10,000 EPS	11,000 EPS	11,500 EPS
50th (median)	0.0 ms	4.0 ms	41.0 ms	487.0ms
75th	5.0 ms	9.0 ms	66.0 ms	715.0 ms
99th	32.0 ms	47.0 ms	126.0 ms	871.0 ms
99.9th	58.0 ms	58.0 ms	135.0 ms	895.0 ms
100th (maximum)	67.0ms	62.0 ms	138.0 ms	895.0 ms

你选择 9 000 事件/秒作为起点，因为它是总事件数量（360 000）的一半。在这个吞吐量下，交易策略的性能和 4 000 事件/秒相比没有发生质变。因为结果在这个吞吐量下看上去还合理，所以你在 9 000 和 12 000 之间继续增加事件量。在 10 000 事件/秒时，性能仍然和 4 000 的情况同质。在中位数和 75 百分位数上可以观测到时延，提示性能开始下降了。接着你把吞吐量增加到 10 000 和 12 000 的中点，也就是 11 000 事件/秒。结果和 4 000 事件/秒相比发生了质变，每个指标都下降了大约一个数量级。无疑这个性能结果已经变差了，但是这解决和 12 000 时类似吗？

你现在心怀警惕，因为 11 000 事件/秒已经大约是 12 000 事件/秒的 90%了，而测试解决却没有 90%的相似度。如果交易策略系统的性能是线性下降的，那么时延应该大约是 12 000 事件/秒时的 90%左右。由于观察到的情况不符，你决定再增加一个吞吐量，11 500 事件/秒。在这个吞吐量级别，你跑了 31 轮测试，总共 356 500 个事件。吞吐量增加了大约 5%，导致观察到的中位数时延增加了大约 11 倍，99 百分位时延增加了大约 6 倍。这结果清晰表明交易策略系统的运行时性能指数是下降的。为了更好理解结果，我们快速地做了下图：



柱状图展现了性能的指数下降。有意思的是，我们观察到所有百分位数的时延都遵循相同的模式，进一步验证了我们的猜测：交易策略系统已经耗尽了处理请求的能力。在开始投入提高交易策略性能前，你仔细考虑，“我要如何界定指数级的时延增长？”



如果不是所有百分位时延都遵循相同模式，假设中位数和 75 百分位数在所有吞吐量下保持相对的稳定，这代表的含义和我们所碰到的情况是一样的吗？花点时间思考什么情况下会造成这样的表现。

无界队列（unbounded queue）的危险

基准测试揭示了一个性能调优上的普遍真理：无界队列是性能杀手。这里我们用队列术语代表一条等待线而不是特定的 queue 数据结构。比如说，这个基准测试把事件存放在一个 List 来作为队列。在生产环境下，在多个层级上存在队列。BboUpdated 的发送方很可能在应用级别把事件排队，接着，在网络协议级别（比如 TCP）可能会用自己的一组队列来管理发送给消费者的数据。当事件处理的速度低于事件产生的速度时，系统变得不稳定，因为待处理工作持续在增长。如果能确保无限内存和 0 响应时间，应用也许能持续处理不断增长的队列内容。然而在实践中，如果系统不能稳定的保持消费能力大于等于事件产生能力，系统迟早会滑出轨道。系统的硬件资源是有限的，当消费者落后时，它会申请更多资源来追赶增长的任务队列。考虑极端场景，增加的内存需求导致更频繁的垃圾回收，反过来进一步降低消费速度。这是一个循环问题，最终会耗尽内存资源，导致系统挂掉。

通过查看交易系统代码，你会发现在交易系统里存在一个消息处理队列。这个应用级队列使用 LinkedBlockingqueue 来隔离网络 I/O 线程和应用线程。在基准测试里，驱动测试的线程直接把事件加到队列里，模拟生产环境下网络线程从外界接到事件的过程。把应用的不同逻辑组件分组到不同的线程池是获取性能和并行处理和常见实践。



前面在探索用 Future 和 Task 做并发时，我们间接使用了队列。从 Future 和 Task 接收提交的 ExecutorService 用一个 BlockingQueue 来管理负载。提供 Executors 的工厂方法不允许调用方提供自己的队列。如果你探索一下这些工厂方法的实现，就能发现创建的 BlockingQueue 的类型和大小。

在网络层和应用层之间加一个缓冲区通常有利于性能。队列是应用能够容忍暂时的消费性能下降和生产者消息爆发。然而，如我们在基准测试里所见，缓冲区是把双刃剑。LinkedBlockingQueue 的默认构造器实际上是无界的，其上限设置为等于最大整数值。

由于无限制的缓存消息，当生产速率持续的大于消费速率时，交易系统的性能降级到了不稳定的状态。

应用背压（back pressure）

如果我们反过来选择把接受事件的队列限制在较小的阈值呢？当生产速率超出消费速率并且队列达到指定容量时，系统的一种选择是阻塞知道队列里有了空档。阻塞迫使事件生产停止，这正是应用背压策略的表现。在这个术语里，压力指待处理的事件队列。压力自我明示了资源使用的增长（比如内存）。通过采用阻塞后续生产的策略，系统把压力反压给生产者。应用级的消费者和生产者之间存在的任何队列最终会达到容量上线，迫使生产者改变生产速率以便继续发送事件。

为了实现背压策略，所有队列都必须限制大小以免过度使用资源，当队列满时，生产者必须停止向队列发送消息。用 JDK 提供的 `BlockingQueue` 接口来实现是很直观的。比如说，下面的代码片段显示了如何用 `LinkedBlockingQue` 来实现这种策略：

```
val queue = new LinkedBlockingQueue[Message](1000)
queue.put(m)
```

在这段代码里，我们构造了一个 `LinkedBlockingQueue`，容量限制为 1 000 消息。基于对生产环境的了解，你觉得保持 1 000 消息的内存队列不会耗尽内存资源。第二行代码演示了把消息加入队列的阻塞操作 `put`。

在应用背压策略时，队列大小选择是很关键的。为了演示这一点为什么很关键，我们假设已经度量到消息从队列里被消费的耗时是 0.5 毫秒。一个时间的处理最大时延是 0.5 毫秒加上在队列里的等待时间。考虑如下场景：队列大小是 1 000，队列里已经有 999 个事件，这时来了一个新事件。在最差情况下，新事件要等待 499.5 毫秒以便其他 999 个已经在队列里的事件处理完，再加 0.5 毫秒的自身处理时间。把队列大小设置为 1 000 意味着最大时延是 500 毫秒，这显示出最大时延和队列大小直接正相关。

更正式的队列大小评估步骤需要考虑环境资源，并且理解业务上能容忍的最大时延。根据和 Dave 的非正式讨论，我们学习到即使几毫秒的延迟也会使一笔好交易变成坏交易。在和 Dave 进一步确认前，我们暂且假定 10 毫秒是能容忍的最大时延。根据这个信息，我们可以计算出一个能确保 10 毫秒时延的队列大小。 拿前面的例子来说，根据以下的最差场景算数来计算：

```
maximum total processing latency = queue size * maximum processing time
```

我们重排这个方程来求解队列大小：

```
queue size = maximum total processing latency / maximum processing time
```

根据这个公式，替换上已知数值来计算队列大小：

```
queue size = 10ms / 0.5ms = 20
```

根据计算，我们需要把队列大小设置为 20 个元素，以便确保在最差情况下也能在 10 毫秒内完成处理。如果想进一步了解背压，我们推荐你阅读 Martin Thompson 的博客：http://mechanical-sympathy.blogspot.com/2012/05/apply-back-pressure-when-overloaded.html。

Martin 是高性能软件开发方面的专家，这篇博客是学习背压知识的无价之宝。

应用负载控制策略

如果消息生产者能够遵循消费者（速率与生产者不同）的规约，不破坏速度慢的消费者，那么背压是一种有效的策略。但是有些场景下无法采用背压策略，尤其是在处理第三方系统时。在这些场景下，我们需要考虑别的策略来提升系统容量而无须对业务逻辑做算法改进。



作者曾在**实时报价**（real-time bidding——RTB）领域工作，报价系统参与拍卖来为显示广告的机会报价。在这个行业，对于报价系统跟不上拍卖速率的容忍度很低。无法及时响应拍卖（出价或不出价）的比例过高会导致报价系统被关进处罚区。在处罚区里，拍卖速度会略有降低。停留在处罚区里的时间达到一定程度会被禁止参与任何拍卖，直到解决了性能问题。

我们重返之前用来讨论背压的场景。应用背压的前提条件是达到队列的容量上限。当队列满时，我们的第一个策略是阻塞进一步添加事件，直到有空档出现。另外一个值得调研的选择是丢弃事件，因为系统已经饱和了。丢弃事件需要更多的领域知识来理解生硬的终止流程在业务上意味着什么。在交易系统领域，交易策略在决定要买或卖时发回一个响应。如果交易系统没有决定要买或卖，那它不需要发回响应。对于交易系统领域，丢弃一

个事件仅仅意味着终止处理。对于别的领域，比如 RTB，丢弃一个事件意味着终止处理并且发回一个消息表示不出价。

另外，每个事件是最佳买入和卖出的一个快照。如果不是快照，假设交易策略系统不是收到 `BboUpdate`，而是收到一系列对最佳买入和卖出的修改的离散事件。这有点类似于基于状态 vs 基于操作的 CRDT。丢弃一个事件意味着只有部分信息，直到接到后续消息。在这种场景下，非常重要的是和领域专家和产品经理讨论来判断信息不完全是否能接受以及能容忍多长时间。

引入负载控制是处理高性能系统时的另一个思维转换。和背压一样，这是另一个机会来重新考虑我们到目前为止对性能提升做出的假设。在聚餐时和 Dave 的讨论提供了极好的洞察，提示我们采用负载控制策略。Dave 指出过时的 `BboUpdate` 事件对交易策略的效益产生的伤害大于收益。我们可以挑战以下两个假设：

- 所有事件都必须被处理
- 正在处理的事件必须被完整处理

我们能够挑战这些假设是因为 Dave 还指出 MV 交易公司值处理高频交易。如果一个 BBO 更新事件被丢弃，Dave 非常确信很快会有下一个事件到来。我们来深入了解一下如何应用这种策略。

拒绝工作

拒绝工作不是叫你拒绝迭代任务，抱歉！当我们在这个上下文讨论工作时，工作这个术语是指处理开销。在刚做了基准测试的交易系统里，工作是指处理新的 BboUpdated 事件。尽管我们还没有深入阅读代码，但是从前面的基准测试可以看出存在一个应用级队列用来接收来自网络的 `BboUpdated` 事件。这个队列是应用的入口，正是我们因为容量原因拒绝接收事件的应用级机会。

根据之前的领域调研，我们学习到想要拒绝一个请求，只需要直接丢弃不做响应就可以了。交易系统只有在希望做交易时才需要给出响应。这意味着当容量满时实现拒绝工作策略只要丢弃消息就可以了。

通过查看交易系统源代码，我们发现架构非常初级。在启动时，构造一个 `LinkedBlockingQueue` 来缓存 `BboUpdated` 事件，然后启动一个消费者线程开始从队

列里消费事件。下面的代码片段展示了这个逻辑：

```
val queue = new LinkedBlockingQueue[(MessageSentTimestamp, BboUpdated)](20)
  val eventThread = new Thread(new Runnable {
    def run(): Unit = while (true) {
      Option(queue.poll(5, TimeUnit.SECONDS)) match {
        case Some((ts, e)) => // process event
        case None => // no event found
      }
    }
  })
  eventThread.setDaemon(true)
  eventThread.start()
```

根据前面的工作，我们看到工作队列大小要设置为 20 以确保最大处理时延小于 10 毫秒。队列初始化之后，创建并启动消费者线程。处理逻辑省略了，不过我们可以观察到这个线程唯一的目的就是在有事件可用时消费事件。把工作加入队列的逻辑很简单。这段代码假设一个 MessageSendTimestamp 和一个 BboUpdated 位于作用域里，也就是 ts 和 e：

```
queue.put((ts, e))
```

在探索背压时已经知道 put 是阻塞调用。因为我们现在的策略是丢弃工作，就不能再继续用 put 了。相反，我们可以使用 offer。如 API 文档里所说，offer 返回一个布尔值，指出元素是否被成功加入了队列。当队列满时，它返回 false。这正是我们想要的语义。我们可以相应地修改代码：

```
queue.offer((ts, e)) match {
  case true => // event enqueued
  case false => // event discarded
}
```



这个模式匹配是一个好引子让我们可以给大家介绍一下内省和透明的应用监控。比如说，跟踪交易系统丢弃的事件数很可能是一个有意思的业务监控指标。这个信息对于数据科学家团队可能也有用，他们可以做线下分析来看丢弃掉的事件数和公司的盈利之间的关系。每当你碰到一个状态改变时，值得考虑一下是否应该记录一个

> 监控指标或者发送一个事件。花点时间考虑你自己的系统里的状态改变。你是否记录了状态改变监控信息用于给非技术团队成员研究？

执行 30 轮 12 000 事件/秒的基准测试，总共 360 000 事件，产出如下结果：

Metric	12,000 EPS with queue size = 20
50th (median) latency	0.0 ms
75th latency	0.0 ms
99th latency	3.0 ms
99.9th latency	11.0 ms
100th (maximum) latency	45.0 ms
Mean latency	0.1 ms
Events processed as percentage of total events	31.49%

这个表格加了两行，记录观察到的平均时延和总共 360 000 事件中实际处理的消息所占的比例。这行很重要，因为系统现在拒绝部分事件了，这是一个用吞吐量换时延的例子。时延数据与前一次相比好极了。最大时延比我们设计的最大时延大 4 倍。这说明我们的性能模型有点乐观了。高的最大时延也许是由于不巧的垃圾回收暂停加上错误地预估了处理时延。尽管如此，最大时延也比第一次基准测试的结果低两个数量级。我们还观察到 99%的请求时延小于等于 11 毫秒，处于我们预期的最大时延 10%的波动范围内。

虽然时延数据看上去非常棒，但吞吐量可就不怎么样了。由于我们新采用的负载控制策略，只有大约 30%的事件被处理了。当事件得到处理时，确实处理得很快，但不行的是 2/3 的时间都被丢弃了。在用负载控制做性能调优时需要注意的另一点是，你很可能需要反复调校来在吞吐量和时延之间取得平衡。查看基准测试结果时，你注意到平均观测到的时延是 0.1 毫秒。下一步，你决定根据平均时延来调整队列大小。通过根据平均时延来调整，意味着你决定牺牲一些时延换取更大的吞吐量。重新计算得到新的队列大小：

```
queue size = maximum total processing latency / maximum processing time =
10ms / 0.1ms = 100
```

调整队列大小后重新跑基准测试，观察到如下结果：

Metric	12,000 EPS with queue size = 100
50th (median) latency	3.0 ms
75th latency	5.0 ms

续表

Metric	12,000 EPS with queue size = 100
99th latency	19.0 ms
99.9th latency	43.0 ms
100th (maximum) latency	163.0 ms
Mean latency	3.9 ms
Events processed as percentage of total events	92.69%

如预期的，时延性能和队列大小 20 时相比有所下降。除了最大时延，其他百分位数据最少增加了一倍时延。好消息是最慢时延没有指数增长。吞吐量数据极大提升。我们观察到超过两倍的吞吐量提升，93%的事件都得到了处理。平均时延比之前的 0.1 毫秒增加了 39 倍。为了比较的目的，平均数反映了中位数和 75 百分位数时延的显著增加。

最后，处于好奇，你尝试在保持队列大小为 100 的前提下加倍吞吐量来做一次测试。交易系统会挂掉吗？能处理全部请求吗？还是会发生一些不一样的事？基准测试结果如下：

Metric	24,000 EPS with queue size = 100
50th (median) latency	7.0 ms
75th latency	8.0 ms
99th latency	23.0 ms
99.9th latency	55.0 ms
100th (maximum) latency	72.0 ms
Mean latency	8.4 ms
Events processed as percentage of total events	44.58%

好消息是交易系统没有挂掉。 它承受住了两倍的吞吐量，而时延数据和 12 000 事件/秒的处于同级别。这显示拒绝工作策略使交易系统面对答复在冲击时的表现大为强固。

提高可靠性和达到可接受的时延的代价是低吞吐量。这些时延揭示了限定队列大小的价值，我们在学习背压控制和拒绝工作策略时学到了这一点。在实施负载控制策略后，仅仅通过调节队列大小，我们就可以观察到差异巨大的结果。这里面还有进一步分析和优化的空间。进一步的分析需要把产品经理拉进来一起权衡吞吐量和时延。非常重要的一点是，尽管负载控制策略的实现依赖于很高的技术素养，它的收益需要用业务价值来度量。

中断昂贵的处理过程

我们可以探索的第二种思路是在处理过程完成前终止它。这是一种确保处理器不被浪费在过时数据上的强大技巧。设想系统从队列里取出一个请求，处理到一半的时候被垃圾回收给阻塞了。如果垃圾回收花了几十毫秒，这个请求就已经过时了，继续处理反而会影响收益。更糟的是，继续处理还可能导致队列里后续的事件也过时。

为了应对这种缺陷，我们可以应用类似于拒绝工作策略的方法——为事件处理设置时间上限。通过在处理开始时打个时间戳，我们可以在一些时间点上去检查处理花费的时间。我们来构造一个例子展示这种做法。考虑如下的处理管道，它可以在给事件打好时间戳并更新了监控指标后运行任意业务逻辑。

```
def pipeline(ts: MessageSentTimestamp, e: Event): Unit = {
    val enriched = enrichEvent(e)
    journalEvent(enriched)
    performPreTradeBalanceChecks(enriched)
    runBusinessLogic(enriched)
  }
}
```

为了避免处理过期的事件，我们可以写类似下面的逻辑：

```
def pipeline(ts: MessageSentTimestamp, e: Event): Unit = {
    if (!hasEventProcessingExpiryExpired(ts)) {
      val enriched = enrichEvent(e)
      if (!hasEventProcessingExpiryExpired(ts)) journalEvent(enriched)
      if (!hasEventProcessingExpiryExpired(ts))
performPreTradeBalanceChecks(enriched)
      if (!hasEventProcessingExpiryExpired(ts)) runBusinessLogic(enriched)
    }
  }
}
```

在这个代码片段里，引入了一个 hasEventProcessingExpiryExpired 方法来基于事件区别处理。方法的实现省略了，不过你可以想象查询系统时间，和已知的允许处理时长比较（比如 5 毫秒）。虽然这种做法可以达到中断过期事件处理的目的，但是现在代码把多方面关注点耦合起来了。即使在这么小的例子里，要跟踪处理步骤都变得比较难了。

这代码的痛点在于业务逻辑和打断过期事件这两个不同的关注点交织在一起。提升可读性的一种方法是把要做什么的描述和如何做的定义分离。在函数式编程里有一个东西，称为 free monad，正是用来做这个的。我们来更深入地看一下 free monad，看看怎么用它来提升交易策略的性能。

Free monad

Monad 及其数学基础——范畴论——是一个大主题，值得专门探索。由于你的迭代明天就结束了，你想在明天就把改进后的交易策略系统提交掉，所以我们就不讨论学术内容，而是专注于 free monad 的实用方面，告诉你如何利用 free monad 来解决真实问题。为了演示 free monad 的威力，我们先展示最终结果，再反过来推导 free monad 的原理。作为开始，我们考虑当交易策略系统接到 `BboUpdated` 事件之后需要执行哪些处理步骤：

```
val enriched = enrichEvent(bboEvent)
journalEvent(enriched)
performPreTradeBalanceChecks(enriched)
val decision = strategy.makeTradingDecision(enriched)
decision.foreach(sendTradingDecision)
```

在交易策略系统做出交易决策前有 3 个步骤。如果交易决策时提交一个买入指令或卖出指令，这个决策被发给股票交易中心。`strategy` 是 `TradingStrategy` 特质的一个实现，如下：

```
trait TradingStrategy {
  def makeTradingDecision(e: BboUpdated): Option[Either[Bid, Offer]]
}
```

接着，我们来看看怎么把这一系列处理步骤翻译成 free monad，同时加上提前终止逻辑。

描述一个程序流程

为了构造新版的交易策略处理管道，我们使用 Scalaz 提供的 free monad 实现，`scalaz.Free`。使用 free monad 结合 DSL 得到的最终结果类似下面这样：

```
val pipeline = for {
   enriched <- StartWith(enrichEvent) within (8 millis) orElse (e =>
     enrichmentFailure(e.ticker))
```

```
    _ <- Step(journalEvent(enriched)) within (9 millis) orElse
      tradeAuthorizationFailure
    _ <- Step(performPreTradeBalanceChecks(enriched)) within (10 millis)
      orElse metricRecordingFailure
    decision <- MakeTradingDecision(enriched)
  } yield decision
```

记得我们第一次尝试实现短路逻辑的代码包含一系列 if 语句。从上面代码可以看到，基于 free monad 的代码不使用 if 语句而是定义成 for 推导。这种做法移除了分支语句，使理解问题变得更容易。就算不看 DSL 的实现，你也很可能已经能够读懂管道的处理逻辑。比如说，你很可能已经猜到如果 `journalEvent` 花了超过 10 毫秒，那么处理流程会终止，`performPreTradeBalanceChecks` 和 `MakeTradingDecision` 都不会被调用。

构造处理管道只是故事的一面。在 for 推导的下面是 free monad。创建一个 free monad 包含两个部分：

- 构造一个程序流程的描述。
- 写一个解释器来执行该描述。

for 推导代表我们对程序流程的描述。描述了 BboUpdated 时间的处理过程同时也定义了执行时长约束。为了执行这个描述，我们必须构造一个解释器。

构造一个解释器

我们的解释器类似这样：

```
def runWithFoldInterpreter(
  recordProcessingLatency: ProcessingLatencyMs => Unit,
  strategy: TradingStrategy,
  ts: MessageSentTimestamp,
  e: BboUpdated): Unit = {
  val (_, decision) = pipeline.free.foldRun(
    PipelineState(ts, strategy, e)) {
    case (state, StartProcessing(whenActive, whenExpired, limitMs)) =>
      state -> (hasProcessingTimeExpired(state.ts, limitMs) match {
        case true => whenExpired(e)
        case false => whenActive(e)
```

```
      })
    case (state, Timed(whenActive, whenExpired, limitMs)) =>
      state -> (hasProcessingTimeExpired(state.ts, limitMs) match {
        case true => whenExpired()
        case false => whenActive()
      })
    case (state, TradingDecision(runStrategy)) =>
      state -> runStrategy(state.strategy)
  }
  decision.fold(logFailure, {
    case Some(order) =>
      sendTradingDecision(order)
      recordProcessingLatency(ProcessingLatencyMs(
        System.currentTimeMillis() - ts.value))
    case None =>
      recordProcessingLatency(ProcessingLatencyMs(
        System.currentTimeMillis() - ts.value))
  })
}
```

foldRun 方法是 Free 提供的，用来执行刚才写的程序流程。类似于 foldLeft 的签名，foldRun 而接受一个代表初始状态的值，一个接受当前状态的柯里化函数和下一步要执行的处理步骤。下一步处理步骤用一个叫作 Thunk 的 ADT 来代表，其包含如下成员：

```
sealed trait Thunk[A]
case class Timed[A](
  whenActive: () => A,
  whenExpired: () => A,
  limit: LimitMs) extends Thunk[A]
case class StartProcessing[A](
  whenActive: BboUpdated => A,
  whenExpired: BboUpdated => A,
  limit: LimitMs) extends Thunk[A]
case class TradingDecision[A](
  makeDecision: TradingStrategy => A) extends Thunk[A]
```

Trunk ADT 定义了可以转发给 free monad 的所有可选操作。我们前面展示的管道是通

过把 Thunk 成员组合起来构造出来的。这个管道把 Thunk 的构造隐藏在 DSL 后面以便避免啰唆，提高可读性。下表是每个处理步骤和 Thunk 的对照关系。

Step DSL	Thunk
StartWith	StartProcessing
Step	Timed
MakeTradingDecision	TradingDecision

回到柯里化 foldRun 函数，我们可以看到解释器通过模式匹配来判断下一个步骤是哪个 Thunk。这组模式匹配就是解释器处理我们的程序流程描述的机制。StartProcessing 和 Timed 使用系统时间和给定的过期时间（LimitMs）来判断应该执行哪个方法。StartProcessing 和 TradingDecision 需要来自外部世界的状态来支持执行。来自工作队列的 BboUpdated 事件必须提供给 StartProcessing，Strategy 必须提供给 TradingDecision 来做出交易决策。

foldRun 的返回值是一个元组，包含一个累计状态（在上面代码里抛弃了）和一个 free monad 的解释结果。pipeline 定义的 Thunk 序列的执行结果是 \/[BboProcessingFailure, Option[Either[Bid,Offer]]]。返回值区分了失败场景——有可能是正常的处理失败，也可能是超时。这些错误用 BboProcessingFailure 类型的 ADT 来代表。返回结果的右侧匹配 TradingStrategy 的正常返回结果，代表 pipeline 的所有步骤全部完成，然后产出一个交易决策。最后一步是对交易决策做折叠来记录管道完成时（也就是\/返回时）的时延，并有条件的发送指令给股票交易所。

你应该从这个例子学到我们是如何把要做的事的描述和怎么做这件事分离的。free monad 让我们可以先创建一个程序描述，然后第二步再来构造一个解释器来执行描述。作为一个具体例子，pipeline 的程序没有把业务逻辑和如何实现提前终止纠缠起来。相反，它只是描述说我们的某些步骤存在时间约束。既然已经构造了能工作的交易策略流水线，再来做一次基准测试看看效果如何。

基准测试新交易策略流水线

对新交易测试流水线运行 12 000 和 24 000 事件/秒的基准测试产出如下结果。结果每一行显示两个值。“/” 前的值是加了提前终止逻辑后的结果，用来和不提前终止的结果比较。

Metric	12,000 EPS with queue size= 100	24,000 EPS with queue size= 100
50th (median) latency	1.0 ms / 3.0 ms	6.0 ms / 7.0 ms
75th latency	3.0 ms / 5.0 ms	7.0 ms / 8.0 ms
99th latency	7.0 ms / 19.0 ms	8.0 ms / 23.0 ms
99.9th latency	10.0 ms / 44.0 ms	16.0 ms / 55.0 ms
100th (maximum) latency	197.0 ms / 163.0 ms	26.0 ms / 72.0 ms
Mean latency	2.0 ms / 3.9 ms	6.0 ms / 8.4 ms
Events processed as percentage of total events	90.43% / 92.69%	36.62% / 44.58%

从时延角度来说，提前终止有明显收益。除了最大时延，提前终止在每个百分位上都产出了更低的时延。比如说，在 12 000 事件/秒的测试中，和不提前终止的版本比，半数的请求只花了 1/3 的时间。在 12 000 事件/秒的测试中，最大时延增加了，这可能提示在提前终止后的垃圾回收。对我们的实现有两个可能的改进点：

- 调用 `performPreTradeBalanceChecks` 后，在调用 `TradingStrategy` 前检查处理花费的时长。
- 在交易决策创建后检查处理时长。

在这两种场景下都可以在处理时长超出阈值时终止处理。因为 free monad 实现提供的关注点分离效果，现在很容易看出这两个处理步骤需要关注。想象一下在业务逻辑和提前终止逻辑交错在一起的代码里要看清问题有多难。

从吞吐量的视角来看，我们看到在两轮测试里吞吐量都下降了，这是由于过期的时间被抛弃导致的。我们再次看到了吞吐量和时延的权衡。我们牺牲了吞吐量来获得更低的时延。在我们的业务场景下是值得的，因为基于过时信息的高吞吐量很可能导致更大的损失。

任务解释器

到目前为止付出的努力显著提高了时延性能，牺牲了吞吐量。有没有可能两全其美呢？如果能低时延高吞吐量就太理想了，不过看上去遥不可及。提高吞吐量的策略之一是引入并发。也许我们能让交易策略并发执行来利用多核硬件的能力。在投入开发之前，你联系了 Gary，你的同事，他帮助你发现了指令簿实现的线性特点。你再次和 Gary 确认 MV 交易系统是否线程安全。你回复了一个笑脸，给我们开了绿灯让我们可以继续把交易策略并行化。

从目前我们对 free monad 的探索中，已经看到了程序描述和解释之间的关系。程序描述——用 Thunk ADT 代表——对解释器是透明的。这句话代表了 free monad 的核心概念，在 Adam Warski 的博客上有精彩阐述(https://softwaremill.com/free-monads/)。free monad 的“free”语义是说 free monad 可以被任意解释。我们将看到这个理念的实际展示——可以把当前的解释器改成一个 Task 解释器。为此，我们必须把 Thunk 映射为 Task。Scalaz 提供了一个特质来表达这种映射，叫作 NaturalTransformation，其类型别名为~>。下面代码展示了如何使用 NaturalTransformation 来把 Thunk 映射到 Task。

```
private def thunkToTask(ps: PipelineState): Thunk ~> Task =
  new (Thunk ~> Task) {
  def apply[B](t: Thunk[B]): Task[B] = t match {
    case StartProcessing(whenActive, whenExpired,
      limitMs) => Task.suspend(
      hasProcessingTimeExpired(ps.ts, limitMs) match {
        case true => Task.now(whenExpired(ps.event))
        case false => Task.now(whenActive(ps.event))
      })
    case Timed(whenActive, whenExpired, limitMs) => Task.suspend(
      hasProcessingTimeExpired(ps.ts, limitMs) match {
        case true => Task.now(whenExpired())
        case false => Task.now(whenActive())
      })
    case TradingDecision(runStrategy) =>
      Task.fork(Task.now(runStrategy(ps.strategy)))
  }
}
```

特质定义了一个方法要实现来把 Thunk 转换为 Task。和前面 foldRun 里的解释器一样，解释器需要相同的状态来提供 BboUpdated 事件，MessageSentTimeStamp 和 TradingStrategy。我们用模式匹配来处理每个 ADT 成员的映射。注意 Task.suspend 的用法，其签名如下：

```
def suspend[A](a: => Task[A]): Task[A]
```

和 Task.now 相反，suspend 推迟参数的运算。这是必需的，因为解释器在调用 hasProcessingTimeExpired 时需要检查系统时钟（一个副作用）。用 suspend 把系

统时钟调用推迟到 Task 运行的时候而不是构造的时候。

第二个有意思的点是在翻译 TradingDecision 时使用了 Task.fork。在这里为交易策略管道引入了并发。转换代码完成了，下一步就是运行解释器了。幸运的是，Free 提供了一个类似于 foldRun 的方法，能接受 NaturalTransformation，方法名字叫作 foldMap。下面代码片段展示了如何用 Task 执行现有的 Thunk 管道：

```
pipeline.free.foldMap(thunkToTask(PipelineState(ts, strategy, event)))
    .unsafePerformAsync {
      case -\/(ex) => logException(ex)
      case \/-(\/-(decision)) =>
        decision.foreach(sendTradingDecision)
        recordProcessingLatency(ProcessingLatencyMs(
          System.currentTimeMillis() - ts.value))
      case \/-(-\/(failure)) => logFailure(failure)
    }
```

调用 foldMap 应用转换，生成 Task。Task 通过 unsafePerformAsync 异步执行。我们对 24 000 事件/秒场景运行一个准测试来和原来的 foldRun 解释器对比：

Metric	24,000 EPS with queue size = 100
50th (median) latency	0.0 ms / 6.0 ms
75th latency	0.0 ms / 7.0 ms
99th latency	4.0 ms / 8.0 ms
99.9th latency	13.0 ms / 16.0 ms
100th (maximum) latency	178.0 ms / 26.0 ms
Mean latency	0.13 ms / 6.0 ms
Events processed as percentage of total events	96.60 % / 36.62%

在一台四核机器上运行 Task 解释器在时延和吞吐量上产生了明显差异。从吞吐量视角来看，几乎所有的时间都能被处理了，远胜于原来的 36%。吞吐量提升显示了通过 Task.fork 来实现运行时并行获得的额外的处理能力。我们同时观察到明显的时延下降，同样要归功于在多核机器上运行 Task.fork。有意思的是，高百分位数上的时延接近。如前面已经注意到的，我们没有在处理流程结束时点上检查事件是否超时。这个基准测试告诉我们，明智地使用 Task 在提升时延性能的同时倍增了吞吐量。通过把交易策略逻辑当作黑盒，只调整其运行策略，取得如此成效令人兴奋。

进一步探索 free monad

我们对 free monad 的探索有意避免深入 monad 的理论基础，而是关注与展示其实用性。使用 free monad，我们向你展示了如何把程序描述和程序执行分离。这让我们可以清晰地引入逻辑来中断过时事件的处理。我们还通过写一个 `Task` 解释器给处理管道加入了并发而不影响其构造。核心业务逻辑保持干净，同时获得了很好的性能提升。我们看到 free monad 在这方面的显著功效。程序的描述是一个值，而解释器负责处理副作用。

到此为止，你能够看到应用这个技巧的好处，但是对于其底层机制仍然一头雾水。对 monad 的全面阐述超出了本书的范畴。通过研究本书的随书源代码以及探索其他的学习材料，你会获得对于如何在自己的系统里应用这一技巧的更深入的理解。我们推荐阅读前面提过的 Adam Warski 的博客以及 Ken Scrambler 的演示（`https://github.com/kenbot/free`）。要更深入地理解 monad，我们推荐你阅读 Paul Chiusano 和 Runar Bjarnason 的 *Functional Programming in Scala*。

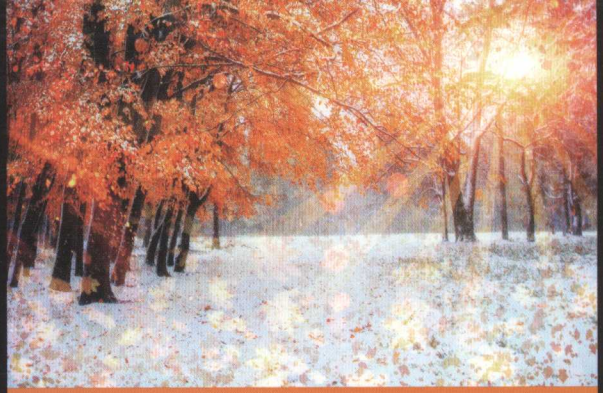
小结

在本章里，我们聚焦于语言无关的高性能系统设计。我们介绍了分布式架构并解释了它们能怎样帮助伸缩系统。我们展示了这种设计范式面对的一些挑战，并且专注于解决在集群里共享状态的问题。我们用 CRDT 来实现高效高性能的节点间同步。通过使用这种数据类型，得以简化我们的架构，并且通过消除专门用于存放共享数据的独立服务来避免产生新的瓶颈。我们还通过避免在关键路径上做远程调用从而保持了低时延。

在本章第二部分，我们分析了队列对时延的影响，以及如何应用负载控制策略来控制时延。通过对交易策略管道做基准测试，我们发现了应用背压和把队列设置为固定尺寸对时延的重要意义。无界队列最终会对生产性能造成灾难后果。对于队列的研究在数学上的正式名称叫作排队论。排队论，和 monad 一样，是一个值得专门学习的主题。我们专注于根据经验观察来驱动改进。学习排队论可以给你更强的理论基础以及为系统性能建立模型的能力。

我们把拒绝工作策略扩展为中断过时事件处理的策略。在这个过程中，我们探索了一个新的函数式编程技巧，叫作 free monad。free monad 让我们可以保持干净的业务逻辑描述，而避免和其运行逻辑纠缠在一起。这种关注点分离的策略让我们可以给处理流程加入并发而不把处理流程搞复杂。我们讨论的这种步骤让你可以写高吞吐量和低时延的系统，在达到容量上限的时候仍然保持强固，同时强调函数式设计。

Broadview
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版



高性能Scala

用Scala和函数式编程范式开发高性能软件

《高性能Scala》将为你武装上创建高性能Scala应用所需的知识。我们从如何定义性能这样的基础知识开始，探索了Scala语言特性及函数式编程技术，同时在所有的主题上都密切关注着性能。

我们虚构了一家名为MV的交易公司，你将作为一个新加入这家公司的软件工程师开始工作。随着你不断学习降低延迟、提高吞吐量的新技术和新方法，你将会把它们应用到解决MV交易公司的业务问题上。学习完本书之后，你将得到充分的知识储备，能使用函数式编程范式来解决真实世界中的问题，并能编写可以用于生产环境的高性能Scala软件。

本书为谁而写

本书假设你具有对Scala编程语言和Java虚拟机的基础认识。你需要能阅读和理解中高级的Scala代码。除此之外无须其他知识。

本书内容要点

- 通过开发JMH基准测试，以及使用Flight Recorder做profiling去分析JVM应用的性能。
- 发掘出Scala语言特性的使用场景及性能折衷的情况，了解急切集合及惰性集合。
- 研究流式处理管道，探索使用事件溯源模式来提升性能。
- 深入异步编程，使用Scala的Future和Scalaz的Task来榨取多核系统的性能。
- 利用最终一致性的优势，结合无冲突复制的数据结构（CRDTs）来设计无须同步的分布式系统。
- 理解队列对系统性能的影响，应用Free monad模式来构建具有高等级吞吐量的健壮的系统。

[PACKT] open source*
PUBLISHING
community experience distilled

上架建议：Java

ISBN 978-7-121-31237-3

9 787121 312373 >

定价：69.00元

责任编辑：张春雨
封面设计：李　玲